January 2009

Wall Street Project

Jessica Lynn Doherty
Worcester Polytechnic Institute

Tanvir Singh Madan
Worcester Polytechnic Institute

Xing Wei
Worcester Polytechnic Institute

Automated Testing and Continuous Integration for the Deutsche Bank Index Quant Team

A Major Qualifying Project
Submitted to the faculty of
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Bachelor of Science

By
Jeccica Doherty
jdoherty@wpi.edu

Tanvir Singh Madan
tmadan@wpi.edu

Xing Wei
wx6872@wpi.edu

Date of Submission: January 7, 2009

Abstract

The goal of this project was to reduce risks from the applications of the Index Quant team at Deutsche Bank. Because their Index Calculator project is frequently updated, it is important to continuously build the project and check for regression errors. Our task was to deploy and use various existing tools to create both a continuous build system and an automated testing system for this project. By using these tools, we created a comprehensive build system to automatically run the build and the regression tests whenever a change is made to the project code.

Acknowledgements

We would like to extend sincere thanks to our sponsors for giving us the opportunity to work in their company. This project could not have been completed without Jason Oh, Jenny Sappington, Nikhil Merchant, and Michel Nematnejad, who have spent a lot of their time assisting us with the project.

Also, we would like to thank our advisors, Professor Donald Brown, and Professor Daniel Dougherty, who have provided us with constant support and valuable advices throughout the project.

Lastly, we would like to thank the people who made this project possible in the first place. They rescued us from jumping on and off from various MQP's! Our sincere thanks to Professor Art Gerstenfeld and Teddy Cho for this project opportunity.

Table of Chapters

Table of Figures

Table of Tables

1 Executive Summary

Software testing is a very important stage in every development lifecycle. The earlier that testing is done, the less time is spent fixing mistakes, and the lower the costs involved in producing the software as a whole. Testing is done in two different ways: unit testing, which tests a specific piece of functionality in the code, and regression testing, which tests that previously working features still work after a new change is made to the code. This project focused on regression testing and catching potential errors immediately after a change is made to the code.

This MQP focused on automating the build and regression testing of Deutsche Bank Index Quant's Index Calculator. The Index Calculator (IC) is used constantly to calculate analytics (unit holdings and level calculations) in relation to various benchmark and tradable indices. Since the IC has undergone and is subject to constant change, validating its functionality is of utmost importance. When the market changes and indices change, it is necessary to ensure that results returned from the Index Calculator are as expected. If the results are not accurate, it will monetarily hurt the clients and ruin Deutsche Bank's reputation as an investment bank.

The goal for this project was to completely automate the build and regression testing process. This consisted of:

- Automating the build process using a tool known as Hudson.
- Creating regression tests using the FitNesse tool and Java.
- Adding functionality to FitNesse to write to the database to self-contain our (and future) test cases.
- Automating these regression tests with Hudson and linking this automation with the build of the Index Calculator itself.

With the accomplishment of these four goals, we developed a self-contained testing system for the Index Calculator used by Deutsche Bank Index Quant. The benefit of this system established is that the testing is self-contained and the builds (including these new tests) are automatically set off each time a change is made to the code. This means that no preexisting data is required to carry out the tests – all the data required to run calculations on the IC can be input through our test mechanism and then

removed as well. This also provides benefit since the testing framework established makes it very easy for the business end team to validate results and approve of the IC prior to deployment.

With all this accomplished, we left Deutsche Bank with some recommendations to expand upon this project, including:

- Expand our FitNesse test suite to test every asset type.
- Use Hudson to build other related projects to further expand the automation.
- Extend the use of these tools to other departments within IT at Deutsche Bank.

Successful action on these recommendations would be extremely beneficial to DB and could serve as a time saver and a risk cutter to all of the bank's software projects.

2 Introduction

The Deutsche Bank Index Quant (DBIQ) team in Wall Street, NY develops and calculates analytics of various benchmark and investible indices. These indices are usually categorized by asset type (Cash, Bond, Interest Rate Swap, etc.) and are calculated and using a common Index Calculator (IC). This IC was also developed at DBIQ and is subject to various change and modifications based on the markets.

Any developer on the DBIQ team may need to modify the IC. Modifying the IC can affect any or all of the indices based on asset type. A developer may make a change to suit his need and not notice that he has caused an error in the system for another unrelated calculation – the owner of which also not knowing that his calculation is no longer correct. These mistakes could be very costly as these indices are used for internal DB customers as well as external agencies to assist in various investment decisions.

As a result, whenever a change is made to the IC, it would be ideal to test and validate all of the impacted indices. To stage such a test, scripts must be loaded to the database for each impacted index among other preparation. As a manual process, this is very time consuming and inefficient. It is also easy to make a mistake or miss a test, defeating the purpose and essentially doing nothing other than wasting time. This led us to the problem we were to solve: the building and testing of the Index Calculator was completely manual and prone to errors.

To attempt to solve this problem, it is important to understand the current infrastructure being used. DBIQ uses an open-source project management tool called Maven[1] to build the IC and its related components. Each part of the calculator has its own configuration file to tell Maven how to build the project. The file specifies goals (such as compile, test, or package), along with which goals are dependent on other goals, and also what file dependencies exist, and so on. Any Maven project can be run from the command line simply by specifying where the configuration file is and which goal to run.

Maven by itself will build the system when a developer tells it to. A developer may start a build when he makes a change and checks his new code in, or he may not, most likely he will not. When somebody does run a new build, if it fails, he will not know who wrote the code that broke it or where said code exists in the project. Trying to figure out where the problem is and fix it can be a long and arduous task and may not even fall into the domain of the person who has to do it.

[1] (JUnit)

As is standard in Java, DBIQ uses the JUnit[2] framework to create test cases for their index calculations. The IC itself and each of the calculators that it calls on each have their own test class. Each class has a variety of tests that use a prior date and other known parameters to ensure that the results are the same as the known value.

The existing JUnit tests can easily be run by any of the developers. Modifying or adding new tests is also a fairly easy but often time consuming task for the developers who work on a specific project. For a developer who is unfamiliar with the code it could be somewhat difficult and even worse for someone who is not a developer and doesn't know Java. This poses another problem – not only are the builds not run whenever they should be, but the business users on the team are unable to understand or write new test cases for the Index Calculator.

Our contribution in an attempt to solve this problem was to implement and build off of various automation tools for building and testing the IC and related calculators. We configured a tool called Hudson[3] to integrate with Maven to provide a continuous build system. Our configuration of this system rebuilds the code every time a developer submits a change, and can be set to notify everyone affected if there is a bug in the code. We used the FitNesse framework[4] to create pages to test previously untested functionality of the Index Calculator. We extended the FitNesse Java classes for our new test cases, and wrote the corresponding wiki pages on our server. These new tests only require a developer for initial creation, and from that point, any user with no Java knowledge or understanding of the system can easily edit or run the tests from a web browser. These implementations will ease the workload of the developers as they farther automate the IC build process. This testing framework also provides great benefit to the business end team which actually develops these indices. They can use these test pages to easily understand and validate the functioning of the IC and hence pass it for deployment purposes.

As a result, these new tools will reduce risks to Deutsche Bank, reduce possible costs associated with these risks, reduce development time and also provide an easy to understand testing framework for both developers and business users to validate results.

[2] (Maven)
[3] (Hudson)
[4] (FitNesse)

3 The Continuous Integration Problem

3.1 Introduction and Problem Definition

As mentioned in Section 2, the DBIQ team wishes to incorporate a tool which provides an easy way to continuously integrate and build the Index Calculator. To do so, we looked into an automated continuous integration tool called Hudson. This section discusses the tool and the steps taken to implement Hudson at DBIQ.

3.2 Background

3.2.1 Hudson Background

Hudson is a tool that provides a continuous integration system. This system makes it easier for developers to integrate changes to their projects and obtain fresh builds. Some advantages that Hudson can bring to a software team are:

- *Continuous building/testing of software projects*: Hudson can repeatedly build or test certain files once users define the appropriate parameters and settings.
- *Monitoring execution of remote projects*: For almost every large project, files are stored on a server and not locally on the developer's machine. Hudson provides an easy interface for anyone to run these builds through a web browser and also provides the results in an extremely easy to understand format.

Along with this, Hudson is simple to initially deploy and understand. It also has some further advantages in the ways in which it can actually carry out the builds:

- *Distributed builds*: Hudson can assign builds to different machines and processors depending on the work load.
- *Permanent links*: once Hudson is established, the users can access the GUI interface at any terminal on the network by using a link provided.

- *JUnit test reporting*: not only is Hudson able to run JUnit tests, but also it logs the errors, build history, and project status on the webpage.

Because Hudson boasts the advantages mentioned above, it has been used by many project teams that seek a convenient way of automated continuous building or testing. Hudson is definitely a great tool for us to use to help achieve our goals. At DBIQ, software programmers have been developing a variety of tools to use in index calculation.These projects are always interrelated due to the nature of financial industry. Through the use of Hudson, we have implemented a tool which allows the Deutsche Bank Index Quant team to test their builds simply through a browser window and to see test resultsin an easily comprehendible way.

3.2.2 Maven Background

At DBIQ, the build process for each project is carried out through Maven. In other words, Maven is the tool used to compile and build each project and to run the corresponding JUnit tests. Although this tool is already used at DBIQ, we wish to provide some insight on it before explaining how Hudson was integrated with Maven to satisfy the need here at DBIQ.

By introducing Project Object Model (POM), Maven is able to control and monitor building, reporting and documentation of certain project. Maven is used by many large project teams all over the world and provides various advantages:

- *Ease and extensibility:* Able to easily work with multiple projects at the same time. Extensible, with the ability to easily write plug-ins in Java or scripting languages. Also, instant access to new features with little or no extra configuration.

- *Model based builds:* Maven is able to build any number of projects into predefined output types such as a JAR, WAR, or distribution based on metadata about the project, without the need to do any scripting in most cases.

- *Coherent site of project information:* Using the same metadata as for the build process, Maven is able to generate a web site or PDF including any documentation you care to add, and adds to that standard reports about the state of development of the project.

- *Release management and distribution publication*: Without much additional configuration, Maven can integrate with your source control system such as CVS and manage the release of a project based on a certain tag. It can also publish this to a distribution location for use by other projects. Maven is able to publish individual outputs such as a JAR, an archive including other dependencies and documentation, or as a source distribution.

- *Dependency management:* Maven encourages the use of a central repository of JARs and other dependencies. Maven comes with a mechanism which allows developers to download any JARs required for building your project from a central JAR repository. This allows users of Maven to reuse JARs across projects and encourages communication between projects to ensure that backward compatibility issues are dealt with.

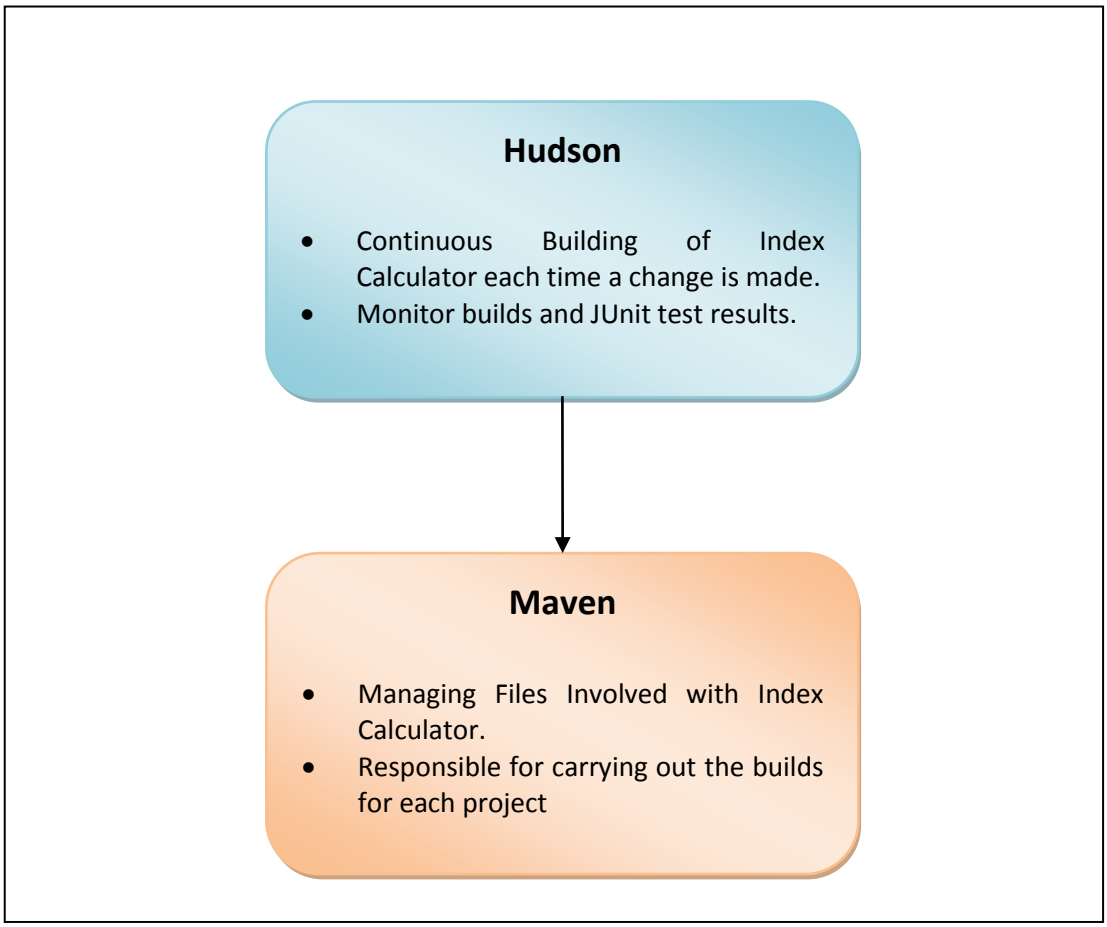


Figure 1 - Hudson integration with Maven

Figure 1 shows how Hudson and Maven can be integrated to develop a continuous build system that can be accessed by any developer.

3.3 Implementation of Hudson

Installing Hudson is straightforward. After copying the open-source archive file to the Linux server, the software is installed through simply executing the downloaded file with Java.

Figure 2 is a screenshot of the homepage of Hudson GUI interface when first deployed. As the image shows, Hudson is very simple to navigate, use, and understand.

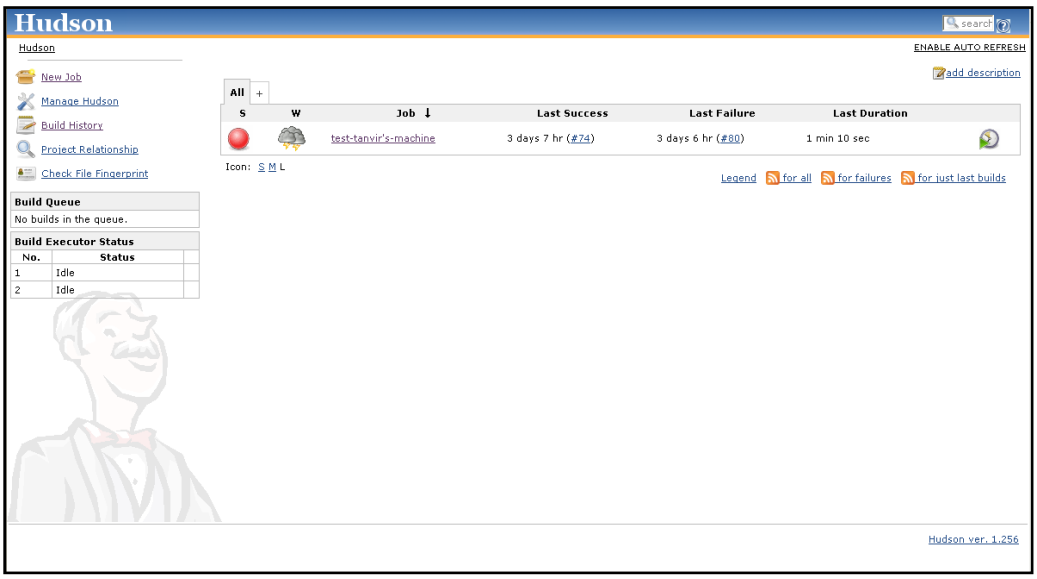


Figure 2 - Hudson interface

To test the functionality of Hudson, we built the projects stored in DB concurrent version system (CVS) repository. We used commands from a Unix shell to understand the repository layout, and then directed Hudson to retrieve and build each of the projects. At the beginning, many builds failed due to the lack of certain project dependencies which Hudson could not find. We manually found and installed each dependency file through Maven to solve this problem. This process resulted in most of the files being successfully built. For those ones that still failed to build, we tried building them without Hudson, and simply ran the build from the Maven build files. We discovered that these failing projects did not successfully build when ran directly with Maven through a Unix shell. Some projects, through either Hudson or the command line, build successfully, but were unstable because the JUnit tests did not pass.

Because we could see that the errors for both the build failures and the JUnit failures were the same, we determined that these errors were beyond the scope of our Hudson deployment and this project.

Since files built in Hudson returned the same result as they were built directly with Maven, we confirmed that Hudson had been successfully established and configured on the server. A total of more than 80 builds were executed throughout this process. Figure 3 illustrates the number of JUnit tests that passed and failed for each build, which are shown in blue and red respectively. The x-axis corresponds to the build no and the y-axis to the count of tests that passed or failed. Blue indicates a passed test and red indicates a failed test. The build no simply corresponding to a particular project being build (For example, build 2 might have to do with an Index Calculator Project and build 3 with some other project). This hence provides an easy way to see all the builds and the corresponding number of JUnit tests that passed/failed for each build.

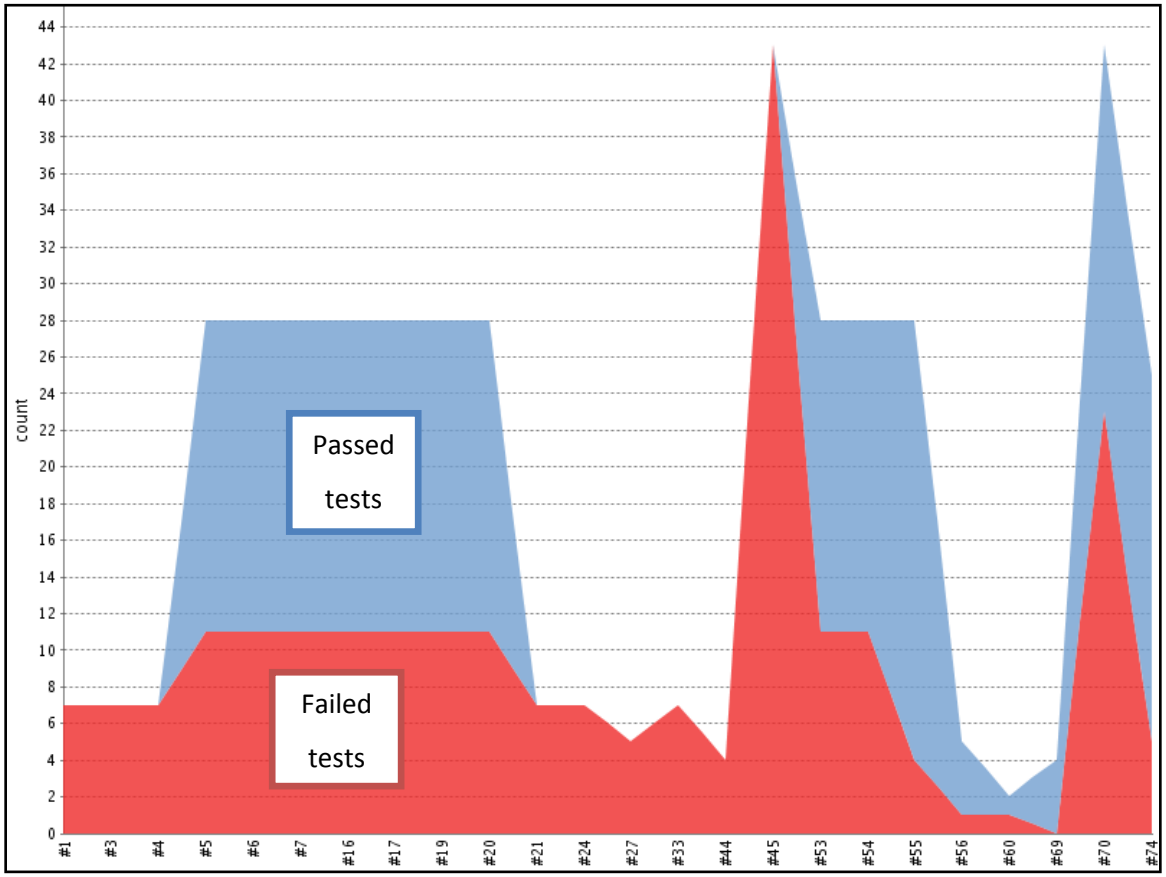


Figure 3 - Test results on Hudson

Looking into the details of the tests that failed and passed showed that they were consistent with the test errors returned by executing only Maven and hence the implementation of Hudson was working successfully.

3.4 Overall Results

We experimented with Hudson to provide the best functionality for DBIQ. In addition to the standard build settings and ability to start a build, we decided to configure Hudson to build whenever a change is made to the repository. We experimented with various settings and ultimately decided to set Hudson to poll the repository every minute to check for changes. With this very short delay between a commit to the repository and the rebuilding of the project, the Index Calculator is rebuilt essentially immediately upon any code changes.

The successful establishment of Hudson enables the Index Quant team at Deutsche Bank to continuously check the functionality of the Index Calculator. Since the results can be viewed by anyone and the build initiated by anyone, the tool is potentially very useful. Since the code is being modified and changed on a daily basis by developers in India, London and the United States, this is a perfect way to keep track of the changes and how they affect the build. Furthermore, it can be applied to a variety of projects with minimal work and hence can be expanded to be used all throughout Deutsche Bank, providing an almost universal build management tool.

4 Automated Testing for the Index Calculator

4.1 Introduction and Problem Definition

Testing problem solutions and ensuring that things work the way they are intended to is universally considered important. In the software industry, the same importance applies. Software testing is a structured investigation carried out to provide developers and users with information about the quality of the product or service under test, with respect to the context in which it is intended to operate. In the financial industry, and specifically under DBIQ, this takes up even more importance as decisions concerning millions of dollars are taken on the basis of the results produced by this team.

Although JUnit tests are a part of the DBIQ testing system, unit tests in general do not fully cover the scope of the software products. Also, with changing times and changing financial conditions, not only are the methods for calculating the indices changing but also the various components that are being considered in this calculation are changing. As a result, there is always some type of change in the system and having an up-to-date regression testing system is of utmost importance to ensure accurate calculations for clients and maintain the bank's reputation.

To get a better understanding of the importance of testing for the DBIQ team and specifically for the Index Calculator, some background information is provided in section 4.2.1. In short, if one index calculation is off, that error will increase exponentially as the index is recalculated and will disrupt the entire system.

4.2 Background

4.2.1 Index Calculator Background

The DBIQ team serves the primary function of the calculation and development on financial indices. Each index is simply a combination or portfolio of different member securities or participants in a way to define an overall representative number.

In general, an index could be of two types[5]–

Benchmarking

A benchmark index should be representative of the market segment that it tracks, and its members should be selected according to well-defined criteria. Examples include the S&P 500, the FTSE 100. Using benchmark indices, we can track how a given market is performing, and compare performance against that of other markets.

Investment strategy

Indices can implement investment strategies, which can be packaged, marketed, and sold to investors. Typical structures include index-linked notes, total return swaps, and ETFs.

The question then arises as to how an index is actually calculated and what are the components that go into it. The specifications of an index are given below [6]:

Members – An index is the sum of its parts. What representative securities can we choose to give an accurate snapshot of market or to yield the greatest returns?

Weights – Do we treat each component equally or give greater importance to some components over others?

Rebalance frequency – How often does member selection occur? DBIQ defines 4 standard rebalance periods: daily, monthly, quarterly, and market triggered.

Return type – Do we express returns in the local currency of its components, or convert to another currency? If we convert, do we account for FX hedging or not? DBIQ defines 3 standard return types: local, hedged, and unhedged.

Currency – In what currency do we express the index and its returns?

[5] (DBIQ Wiki/Blog)
[6] (DBIQ Wiki/Blog)

Calculation type – How are we calculating returns? The 3 possibilities are: ER (excess return), TR (total return), and PR (price return). The calc type of an index is largely dependent on the type of assets that compose it.

Price group – Multiple price sources can exist for the same security, which source shall we use as basis for index calculation?

The old DBIQ Index Calculator project was based on multiple types of index calculation. This made the support of indices for research, trading, structuring, legal and back office more complicated as many indices have different rules and calculation methodologies. These exceptions also increase the development time for each index as new legal documents and calculation code are required.

As a result of this code structure, a different type of formulation is required to calculate the index for each different asset class that the index has in its member list. Because of the number of asset types and of the ever so frequent changes in the financial sector, developing new indices or asset classes was a lengthy and difficult process with the old system.

This system has changed and now DBIQ makes the use of a standard index calculator. The structure of the new calculator is shown below in Figure 4. As the figure shows, input parameters such as the index ID and calculation type are passed to the singular Index Calculator. The IC retrieves the necessary data from the database and performs the calculation based on the input parameters. This new calculator has enabled greater focus on the design of new indices. The lead time for creating indices in DBIQ has been reduced, as well as the process for writing legal index descriptions. Although the team decided that this is the way to write the software, a lot of work is still to be done and this project aims to help the team do it correctly.

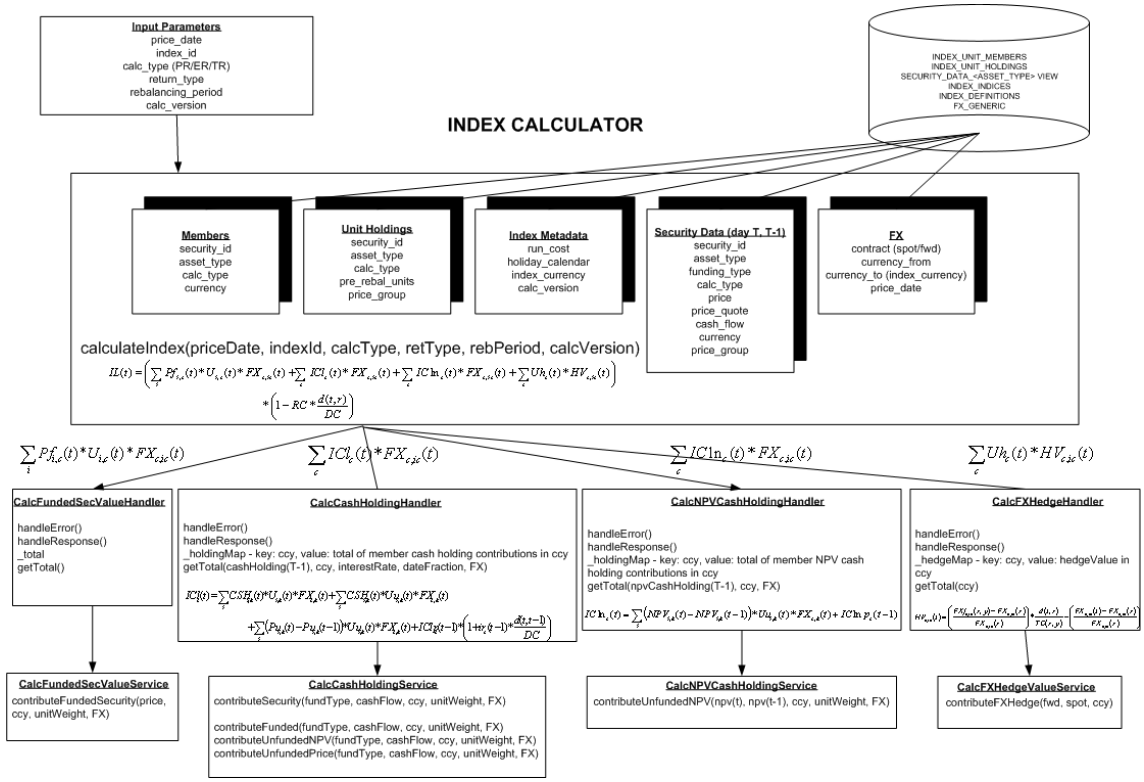


Figure 4[7] – DBIQ Index Calculator

The purpose of this project was to implement an automated testing tool for the new Index Calculatorin a way such that test results can be seen and understood by business end users. It is seen as a requirement in the larger picture development of this new index calculator and as an extremely useful tool both for future developers and for future business users.

4.2.2 Testing Tools Background

Automated testing has both advantages and disadvantages. It reduces and even eliminates human error. Or, if the automated tests are poorly written, it ensures human error. It saves time – eventually, after making up for lost time spent simply preparing the automation. Despite the disadvantages, automated testing can usually save a company very much time and money over manual testing[8]. This is especially true for a system such as the one at Deutsche Bank, where one change to the Index Calculator affects

[7] (DBIQ Index Calculator)
[8] (Automated Testing)

nearly all of the indices. When this system is fully automated, any change to the IC will set the tests to run, and a result will be returned to the users with no prompt. Ideally, any errors that this change causes will be reported to the person who made the change and any users that it affects.

There are a variety of tools to use for Java acceptance tests. One such tool is the FitNesse Acceptance Testing Framework, which we decided to use for the Index Calculator tests. FitNesse is a completely automated, web-based wiki-style testing framework. From the business end,it only requires an administrator to deploy the software, and then any user can update the wiki using a simple table format. On the development end, it takes Java knowledge and understanding of the code to create the back end fixtures for the test pages that anyone can use.Each page marked as a test page has a test button, and any user can run the tests and see the green = pass, red = fail results.[9]

To begin this project, a team member in London provided us with a test case in FitNesse and our task was to deploy it and get a feel of how it works. Using her Java fixture we created the wiki pages to run the tests on our specific data. This was a simple task, as it did not require new Java code, but it gave us a helpful understanding of both FitNesse and the Index Calculator. Once this task was completed, we were also asked to look into some other testing tools – see Appendix A for details. We explored the documentation and began deployment of each to test their usefulness, but FitNesse stood out as the clear winner among these tools. Once we decided to continue with FitNesse, our next step was to write and implement our own test cases for the Index Calculator using FitNesse.

4.3 Implementation of FitNesse

4.3.1 Initial Deployment of FitNesse

Our first FitNesse fixture to write on our own in Java was a type of row fixture to test the Security Holdings calculator functionality. A row fixture is a FitNesse test table style designed for checking query results. An array stores each of the rows of the table, each row being one instance of a Java object. FitNesse compares the input from the table on the wiki page to the actual results returned by the Java query. It checks for and displays any differences in the fields for an object, displays any entries that the

[9] (FitNesse)

query returned that are not specified in the table, and marks any entries specified in the table that were not returned as missing.

The first test that we wrote deals with index rebalancing. Index rebalancing occurs periodically to better represent the market state or to stop relying on securities that are not performing well. In each rebalancing period, every security in the index has a unit value, the index holding of that security for the period. Each security has a price as of any given date, and an asking price as of the same date if available. Every security also has a current percentage weight, the percentage of the index that it currently represents, and a target percentage weight, the percentage of the index that it will represent after rebalancing.

When the Security Holdings Calculator is run because of index rebalancing, it changes the unit value and the percentage weight for each affected security. These new values are stored in the database along with the rest of the information about the security. Our job was to write a new row fixture type and corresponding wiki table to ensure that these calculations are done correctly. The numbers to test the fixture came from a preexisting JUnit test class and from queries to the database itself.

We wrote a new Java fixture that delivers the query results from the database to FitNesse. This class used the input data from the wiki table to call the various methods to get access to the database and extract the results from there. Once this is done, each object instance is sent to FitNesse. To write the fixture, we used two Java classes using the Adapter design pattern. The main fixture class carried out the queries and the calls to the database. The secondary class was the adapter class, and the object type returned by the query. Using two classes makes it very clear to any future developers the exact relation of the returned object to the wiki test table. The results of this FitNesse test are shown in Figure 5.

preUnits	postUnits	currency	price	askPrice	percentCurrentWeight
0 *expected* 3.98838E-4 *actual*	0	EUR	1	1	0 *expected* 2.82488E-4 *actual*
0.028930755	0.025540553	EUR	94.87059699	95.50422699	1.94399417
0.0274971	0	EUR	85.01259205	86.47241205	1.655669762
0.03771105	0.033445769	EUR	93.75501178	94.68314178	2.504186565
0.029806679	0	EUR	97.26901493	97.89401493	2.053485636
0.06810907	0.064707741	EUR	98.39606425	98.60888425	4.746639362
0.025784618	0.023236834	EUR	92.00836425	92.64448425	1.680318374
0.033671968	0.030110297	EUR	94.6733963	95.2022563	2.257875616

Figure 5 – First test results on FitNesse

This screenshot shows the columns represented as fields in the object and also from the database – preUnits, postUnits, currency, price, askPrice and percentCurrentWeight. When the value in the corresponding field of the object matches the value in the table, the cell turns green. When the results do not match, the cell turns red and displays the value it obtained from the query as well as the value we had expected it to obtain.

4.3.2 Modifying Fixtures – calculating the 'delta'

4.3.2.1 Problem and Background

As seen above, FitNesse compares a value given in the test page with the value that the code generates. To tell it what to do, we need to pass in values that the results are compared to through the test page to FitNesse. This is shown below:

```
!|com.db.dbiq.calc.index.xa.fitnesse.FitUnitHoldingsRowFixture|${PRICE_DATE}|${INDEX_ID}|
|securityId|assetType|preUnits|postUnits|
|25|CASH|137.7921524807846|137.7921524807846|
|26|CASH_NPV|-2.4469130437537734|-2.4469130437537734|
|27248|IRS|137.707846|null|
```

Figure 6 – Wiki style inputs on FitNesse

Figure 6 shows that FitNesse is calling the FitUnitHoldingsRowFixture Java class and passing in the input parameters PRICE_DATE and INDEX_ID (defined above the table in the same wiki page). The results being checked are securityId, assetType, preUnits and postUnits. The values entered in the columns are what we expect FitNesse to return and are compared against the values generated by the code.

As stated above, if the two values match, the cell containing this value turns green, else it turns red to indicate an error. Each time a cell turns red, FitNesse splits the cell into two cells – one indicating the result it obtained from the code and the other showing the expected result it was checking with. An example is shown in Figure 7.

com.db.dbiq.calc.index.xa.fitnesse.FitUnitHoldingsRowFixture	20080107	10690	
securityId	assetType	preUnits	postUnits
25	CASH	137.7921524807846	137.7921524807846
26	CASH_NPV	-2.4469130437537734	-2.4469130437537734
27248	IRS	137.707846 *expected* 137.7921524807846 *actual*	null

Figure 7 – FitNesse example with a numerical discrepancy

We decided to change this functionality, and modify FitNesse to show the magnitude (delta) of this difference when the values being compared are numerical. Showing this delta would make it easier to see and understand the error and give them further insight as to why the error might be occurring. Sometimes, these indices and their corresponding securities have attributes which vary very little with time and hence catching even the smallest error is essential to understand what is going wrong.

4.3.2.2 Proposed Solutions

In order to propose a solution to this magnitude issue, we needed to completely understand what happens when FitNesse finds an error and how it decides to deal with said error. To gain this understanding, we looked at the code behind FitNesse. After thorough examination, we found out that FitNesse carries out a query on the information given in the wiki table and then once that information is returned, FitNesse stores it all in an object and then interprets it row by row, interpreting each cell in a row before moving onto the next. When FitNesse checks a cell, if it finds a discrepancy between what was obtained and what was expected it calls on another function to split the cell into 'expected' and 'actual' and displays those results.

- The first solution we thought of was to override this function so that it can calculate the difference, 'delta', and display it on this cell as well as the 'expected' and 'actual' values. The results of this are displayed in Figure 8.

20070831	14083	TR	LOCAL	QUARTERLY	STD
preUnits	postUnits	currency	price	askPrice	percentCurrentWeight
0 *expected* 3.98838E-4 *actual* 3.98838E-4 *delta*	0	EUR	1	1	0 *expected* 2.82488E-4 *actual* 2.82488E-4 *delta*
0.028930755	0.025540553	EUR	94.87059699	95.50422699	1.94399417
0.0274971	0	EUR	85.01259205	86.47241205	1.655669762
0.03771105	0.033445769	EUR	93.75501178	94.68314178	2.504186565

Figure 8 – FitNesse with delta in the same cell

Although this was found to be a functional solution, the DBIQ team preferred a solution where we could generate another column to display the differences. We learned a lot about how FitNesse works by investigating this request. We found out that the function to split this cell into the expected and actual is only called once the query from our code is returned and hence we cannot change the rows returned after this function is called. We tried creating a new class to run the first query, check the actual and expected values, and then display the new data within a query but this issue remained the same – the delta could not be calculated before the new query returned to FitNesse. The problem with this is that

still could not create another column because the functions to split the cells still could not be called until after the table was generated.

- We finally came up with another solution. Investigation of the FitNesse code led us to realize that if a column is left blank, FitNesse has a function to fill it up with the corresponding data - but does not check it because there is no value to check it against. We found the function that does this and decided to override it the same way that we had overridden the other functions. We changed it so that instead of displayed the result it normally would, it displays the latest delta it just calculated. This led us to making another change to ensure that if the actual and expected values matched, the delta value would be reset to zero. This new solution is shown below.

As shown in Figure 9, the delta value is displayed every time a discrepancy is found.

preUnits	delta	postUnits	delta	currency
0 *expected* 3.98838E-4 *actual*	3.98838E-4	0		EUR
0.028930755		0.025540553		EUR
0.0274971		0		EUR
0.03771105		0.033445769		EUR
0.029806679		0		EUR
0.06810907		0.064707741		EUR
0.025784618		0.023236834		EUR
0.033671968		0.030110297		EUR
0.026383249		0.02 *expected* 0.023107142 *actual*	0.0031071420000000002	EUR

Figure 9 – FitNesse with delta is a new adjacent column

Shown above in Figure 9 is the second solution that we obtained by over-riding some methods. As the figure shows, for each column of numerical information, a corresponding delta column is shown on the side whether it is used or not. Also, the delta value is reset so that other blank cells can be filled in with their proper values. The DBIQ team decided that this was indeed the solution they preferred and this solution has been fully implemented.

4.3.2.3 The Permanently Implemented Solution

As shown above, this solution worked as expected. However, we considered the fact that it would only work with this one test fixture the way that we wrote it. We would need to override the same methods in every new fixture class to keep this functionality. Leaving our solution like this would violate basic object oriented principles such as reusability and commonality. Therefore, we decided to make the ability extendable and created two new classes: DeltaRowFixture (extends RowFixture) and DeltaColumnFixture (extends ColumnFixture). RowFixture and ColumnFixture are simply the standard classes provided by FitNesse generally extended to make new fixtures. Any new fixture class that we want to show a delta should extend one of these new fixtures. If this functionality is not desired, developers can continue to extend the standard classes, or simply not put a 'delta' column in the wiki tables. For ease of use for the business users, developers should extend the new classes and leave it to the business users to decide if they wish to display the delta values or not.

4.3.3 Test Suites

The overall goal of this project was to increase automation to save time and effort. While having a FitNesse server filled with pages of tests makes the tests significantly easier to understand, it does not make the job much quicker than writing the tests entirely in Java. To solve this problem, we created a FitNesse Test Suite. As the name implies, a FitNesse Test Suite is a collection of FitNesse test pages. The suite consists of simple links to the wiki test pages that reside under it hierarchically and runs every page under it marked as a test page. As shown in Figure 10, the test suite displays all of the test results on one page so that the user can see everything at once.

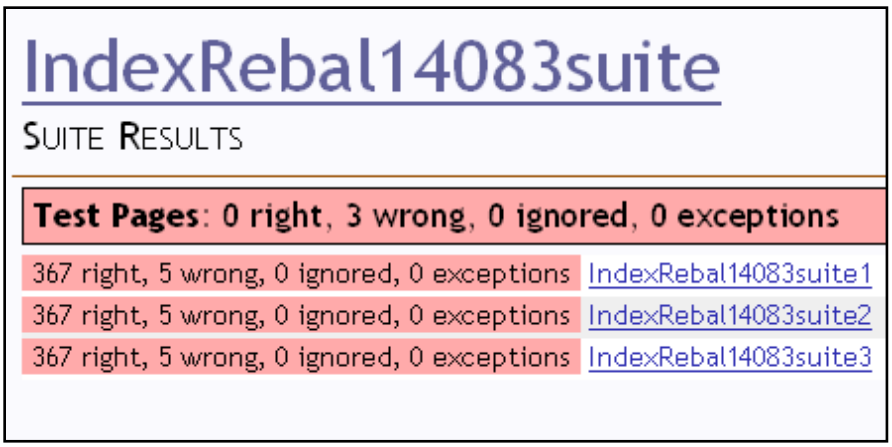


Figure 10 - A snapshot from a test suite showing the results from 3 tests

The final goal would be to have one test suite which when called on will run all the tests under it, which will consist of more suites, ordered by asset type, which store the test cases.

4.4 Self-contained testing

4.4.1 The purpose

Information for our FitNesse test pages came directly from the DBIQ development database (with some changes to our tables be incorrect values and to ensure that these incorrect values were displayed properly). Similarly, our initial fixture code retrieved the same information from the same development database. Thus, preparing our test cases in this manner will always return 100% true results, and does not actually test any new changes to the calculator. We want the tests to be able to call on the Index Calculator to generate the required data and then store that in a database. Our test should then check against that database to make sure the results are correct. With this as our goal, we ventured into possible avenues through which this could be achieved.

4.4.2 Proposals/Solution

To solve this problem, there were a few things we needed to keep in mind. First, we needed to have our test fixtures actually call on the Index Calculator and tell it to carry out calculations rather than just call for the answers on the database. Secondly, we had to move to a different database. The existing

database that we were using already had all of the (theoretically) correct data from previous tests of the IC and hence we could easily alter data used by other developers.

We solved the first problem fairly easily. In the fixtures that were provided to guide us in writing our own, there was an example of a test which calls upon the Index Calculator before obtaining the results from the database. For our tests to do the same, we simply had to follow the model, ensuring that we were calling the correct functions from the Index Calculator, to add in the calls to run the IC and save the results to the database.

Along with this, the underlying input data that is required to carry out the calculations in the first place need to be defined in the new database too. Since generally accepted coding standards encourage self-contained tests, we wanted our tests to insert this underlying data into the test database, carry out the tests and then delete this data.

Keeping this goal in mind, we decided to set a set-up page for each test to insert the required data, then the tests to be carried out and then a tear-down page to remove the inserted data.

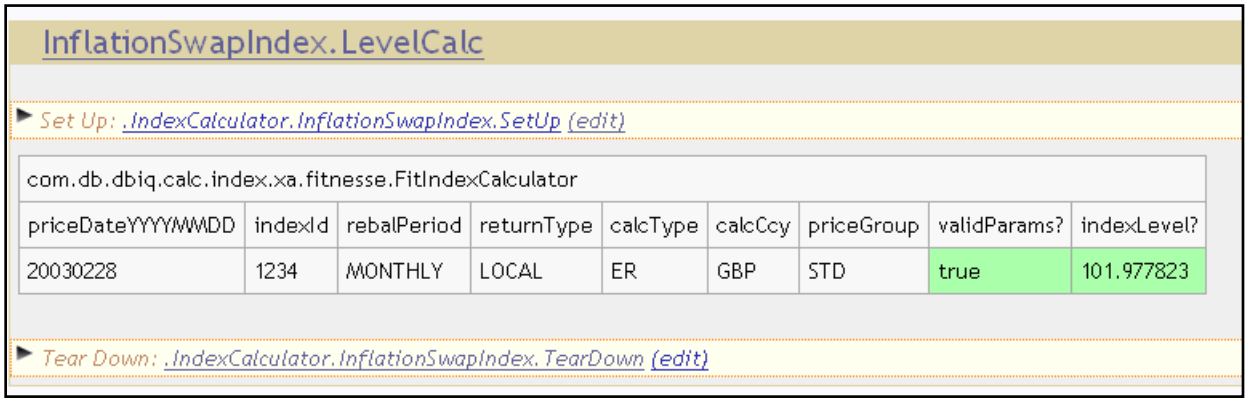


Figure 11 - Set-ups and Tear-downs in FitNesse

As Figure 11 shows, a set up and tear down page can (and should) be used for each FitNesse test to make the necessary changes to the database.

4.4.3 Implementation

Our first solution involved a single DatabaseWriter class. This was a new fixture that allowed a user to enter a table name, the column names to insert to, and any number of rows corresponding to those columns. Writing the class was a fairly complex task. We overrode some of the standard fixture methods

to ensure that the Java code kept track of which row of the wiki table it was interpreting. We created methods to store an array of column names, as well as an array of array of values – this ensured that we could insert as many rows as necessary. We then created methods to write the values of these arrays to the database once the entire table was read.

This DatabaseWriter class was a rough solution that worked but was not very effective. For one thing, it didn't conform to FitNesse standards. The column names were not represented as fields of the class, and the values only corresponded in that they were at the same index in the Java lists created. Another issue with this method was very prone to user error, where any mistakes in the column names or values would not be easy to find.

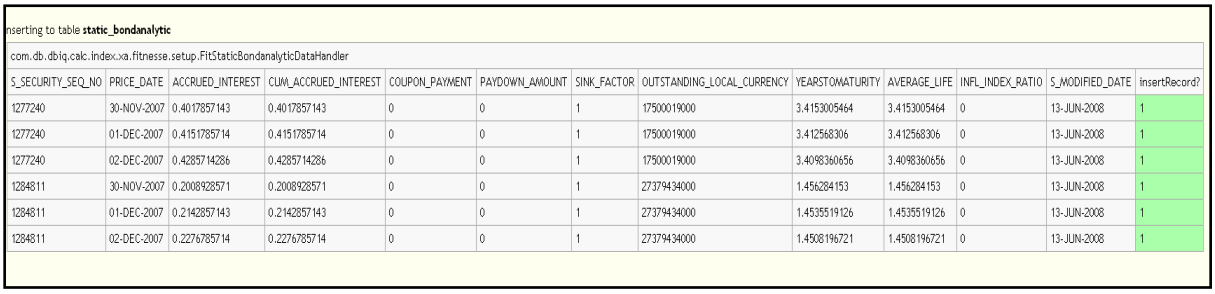
DBIQ Index Calculator - Use this to insert data into a specific index table in the DB	
WriteToindexindices	*This can be used to insert queries to the table 'index_indices'*
WriteToindexlevelmetadata	*This can be used to insert queries to the table 'index_level_metadata'*
WriteToindexrundata	*This can be used to insert queries to the table 'index_run_data'*
WriteToindexreturn	*This can be used to insert queries to the table 'indexreturn'*
WriteToindexrebalancing	*This can be used to insert queries to the table 'index_rebalancing'*
WriteToindexdefinedweights	*This can be used to insert queries to the table 'index_defined_weights'*

Figure 12 - Writing to tables from FitNesse

Figure 12 shows our first test suite that used our DatabaseWriter to write to different tables required for the underlying data to carry out the calculations.

The next solution for the database writing problem came with much help with a DBIQ-London team member. We were given a set of sample Java classes for one particular table, index_indices. Looking at these classes, and the data access class that went with it, we saw exactly how to turn a row of a FitNesse table into a database entry. We then moved our test cases onto the test database and began writing similar fixtures and functions to allow write access to any of the database tables we were using. We had to ensure that for each table to insert into, we had the right columns and properly converted the datatypes (for example, a String in the wiki table that had to turn into a Deutsche Bank-specific enum type). We also had to create a new method in our data access object with the SQL code to insert into that specific table.

As shown in Figure 13 and Figure 14, the insertRecord() function returns 1 (thus showing green) if the record is successfully inserted, and returns 0 (and displays red) if the record is not successfully inserted.

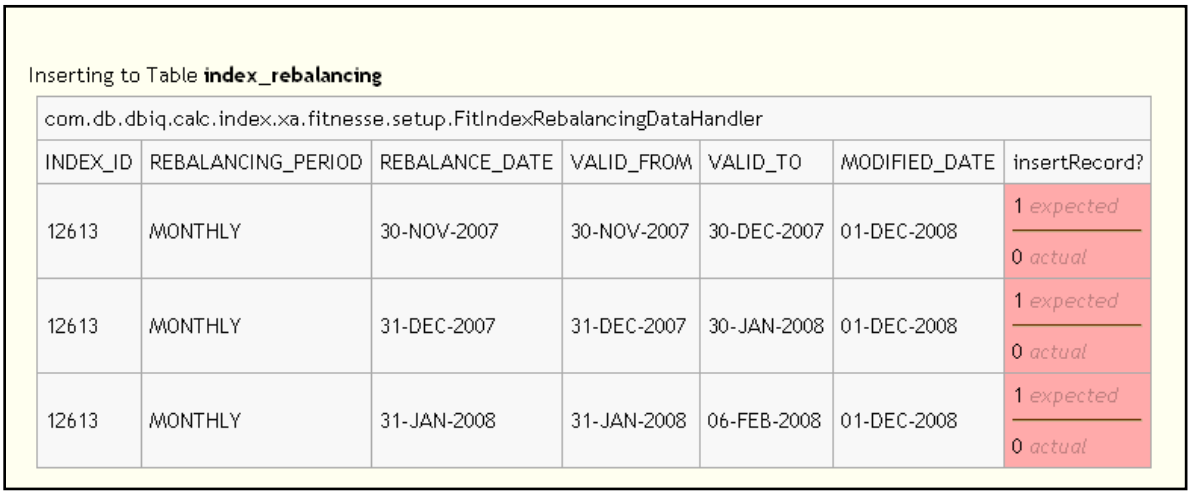


inserting to table **static_bondanalytic**

com.db.dbiq.calc.index.xa.fitnesse.setup.FitStaticBondanalyticDataHandler

S_SECURITY_SEQ_NO	PRICE_DATE	ACCRUED_INTEREST	CUM_ACCRUED_INTEREST	COUPON_PAYMENT	PAYDOWN_AMOUNT	SINK_FACTOR	OUTSTANDING_LOCAL_CURRENCY	YEARSTOMATURITY	AVERAGE_LIFE	INFL_INDEX_RATIO	S_MODIFIED_DATE	insertRecord?
1277240	30-NOV-2007	0.4017857143	0.4017857143	0	0	1	17500019000	3.4153005464	3.4153005464	0	13-JUN-2008	1
1277240	01-DEC-2007	0.4151785714	0.4151785714	0	0	1	17500019000	3.412568306	3.412568306	0	13-JUN-2008	1
1277240	02-DEC-2007	0.4285714286	0.4285714286	0	0	1	17500019000	3.4098360656	3.4098360656	0	13-JUN-2008	1
1284811	30-NOV-2007	0.2008928571	0.2008928571	0	0	1	27379434000	1.456284153	1.456284153	0	13-JUN-2008	1
1284811	01-DEC-2007	0.2142857143	0.2142857143	0	0	1	27379434000	1.4535519126	1.4535519126	0	13-JUN-2008	1
1284811	02-DEC-2007	0.2276785714	0.2276785714	0	0	1	27379434000	1.4508196721	1.4508196721	0	13-JUN-2008	1

Figure 13 - A table of successfully inserted rows



Inserting to Table **index_rebalancing**

com.db.dbiq.calc.index.xa.fitnesse.setup.FitIndexRebalancingDataHandler

INDEX_ID	REBALANCING_PERIOD	REBALANCE_DATE	VALID_FROM	VALID_TO	MODIFIED_DATE	insertRecord?
12613	MONTHLY	30-NOV-2007	30-NOV-2007	30-DEC-2007	01-DEC-2008	1 *expected* / 0 *actual*
12613	MONTHLY	31-DEC-2007	31-DEC-2007	30-JAN-2008	01-DEC-2008	1 *expected* / 0 *actual*
12613	MONTHLY	31-JAN-2008	31-JAN-2008	06-FEB-2008	01-DEC-2008	1 *expected* / 0 *actual*

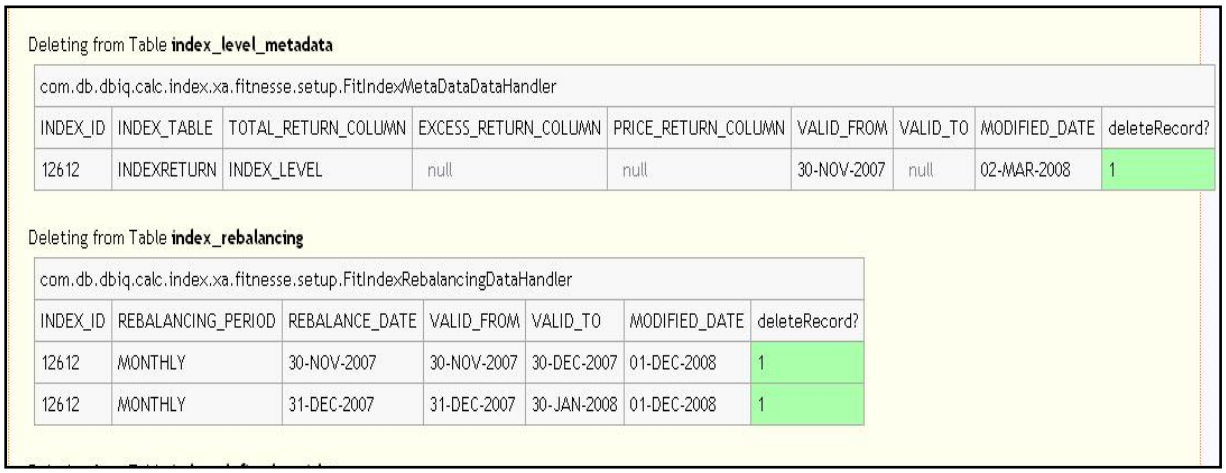
Figure 14 - A table of rows that did not insert successfully.

The task of creating all of the insert tables for our test cases proved to be more difficult than expected. Since our test cases had already worked on the development database, we knew that the cases themselves were correct. However, errors gave us no indication as to whether our fixtures were buggy or if we were missing a table of information, and which table that was. We realized that we had to separate the two problems, and removed our FitNesse insert tables. We directly accessed the database to test whether we had all of the necessary information before moving on and eventually got all of our test cases working.

If we were to use this solution as-is and insert the necessary data, run the test, and then leave it that way, our tests would be far from self-contained. It is possible that by running this way, a test could

return correct (or incorrect) based only on manipulations from a previous test. To fully self-contain the test cases, it is necessary to have a tear-down function after the test that is very similar to the set-up function before the test.

The tear-down functions were significantly easier to write because we had the set-up functions to base them off of and reused the fixtures from the insert tables. We used wiki tables that were almost identical to the set-up tables, the only difference being that they called a deleteRecord() function at the end rather than the insertRecord() function. The delete functions that are used need only the primary key for each table, but the objects are designed to accept all fields. Many of the columns are not necessary for the delete statements, however, we decided that it would be easier to extend these tests in the future if users know that they can just write one table for both purposes. Successful database deletes are shown in Figure 15.



Deleting from Table **index_level_metadata**

com.db.dbiq.calc.index.xa.fitnesse.setup.FitIndexMetaDataDataHandler

INDEX_ID	INDEX_TABLE	TOTAL_RETURN_COLUMN	EXCESS_RETURN_COLUMN	PRICE_RETURN_COLUMN	VALID_FROM	VALID_TO	MODIFIED_DATE	deleteRecord?
12612	INDEXRETURN	INDEX_LEVEL	null	null	30-NOV-2007	null	02-MAR-2008	1

Deleting from Table **index_rebalancing**

com.db.dbiq.calc.index.xa.fitnesse.setup.FitIndexRebalancingDataHandler

INDEX_ID	REBALANCING_PERIOD	REBALANCE_DATE	VALID_FROM	VALID_TO	MODIFIED_DATE	deleteRecord?
12612	MONTHLY	30-NOV-2007	30-NOV-2007	30-DEC-2007	01-DEC-2008	1
12612	MONTHLY	31-DEC-2007	31-DEC-2007	30-JAN-2008	01-DEC-2008	1

Figure 15 – Successful database deletes

4.5 Additional Testing

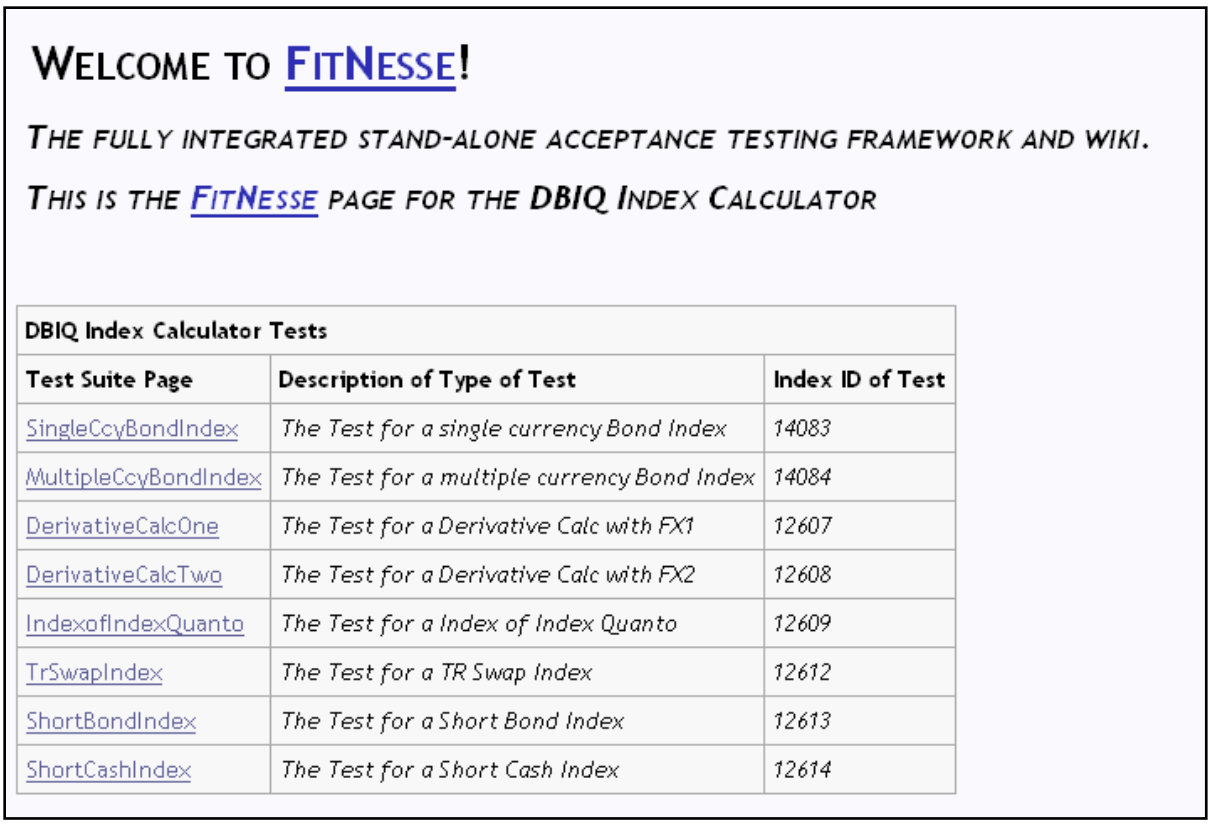
Deploying FitNesse and writing self-contained tests was a significant step towards properly testing the Index Calculator. However, our first stage of writing a test suite didn't cover the IC very well. Our first suite only covered two asset types, and only tested the indices on rebalancing dates (dates in which the securities in the index change weights based on the current market state). We needed to cover about

twenty asset types, and both rebalancing and non-rebalancing dates, so extending our testing framework was a must.

The first step to extending our tests was adding in a non-rebalancing date. This was relatively simple; we just had to look up information about indices that we had already worked with for a different date. We also had to ensure that the calculator ran on every date between the rebalancing date and test date to populate the tables, so we decided to use the day after the rebalance for most of these tests. Because the same type of information was returned as the tests for rebalancing dates, we did not have to create any additional fixtures to do these tests.

Extending the test suite to include every asset type that uses the IC was a significantly larger task. The database was missing some of the information to be tested, so we needed to recreate that data based on index IDs that we were given and previous data for those indices. Then, based on the asset type of each index we either decided to reuse a previous fixture (if it provided all of the necessary data) or wrote new fixtures to test the indices. Adding our new tests to complete the test suite covers the Index Calculator to ensure that any problems made by a change to the code will be caught and able to be easily fixed.

To recreate the missing data, we looked in provided excel spreadsheets that were originally used to generate these indices to find the missing information and appropriately insert them into the required tables. This task was not too straightforward as it required understanding how the calculations were working for these different types of indices and how that translated into the data required in the underlying tables. However, once we looked up the first case and understood how it was calculated, it was much easier to decide what information was needed for each of the subsequent test cases.



WELCOME TO FITNESSE!

THE FULLY INTEGRATED STAND-ALONE ACCEPTANCE TESTING FRAMEWORK AND WIKI.

THIS IS THE FITNESSE PAGE FOR THE DBIQ INDEX CALCULATOR

DBIQ Index Calculator Tests		
Test Suite Page	**Description of Type of Test**	**Index ID of Test**
SingleCcyBondIndex	*The Test for a single currency Bond Index*	*14083*
MultipleCcyBondIndex	*The Test for a multiple currency Bond Index*	*14084*
DerivativeCalcOne	*The Test for a Derivative Calc with FX1*	*12607*
DerivativeCalcTwo	*The Test for a Derivative Calc with FX2*	*12608*
IndexofIndexQuanto	*The Test for a Index of Index Quanto*	*12609*
TrSwapIndex	*The Test for a TR Swap Index*	*12612*
ShortBondIndex	*The Test for a Short Bond Index*	*12613*
ShortCashIndex	*The Test for a Short Cash Index*	*12614*

Figure 16 - Tests for the various asset types

Figure 16 shows how the page started to get organized once the number of asset classes that we tested started growing. We provided a description of the test taking place and the asset type it dealt with. This page itself can be run as a test suite, and each of these links actually contained a test suite as shown in Figure 17. Each inner test suite contains a test for the index level on a non-rebalancing date and another for the units of each underlying security on a rebalancing date.

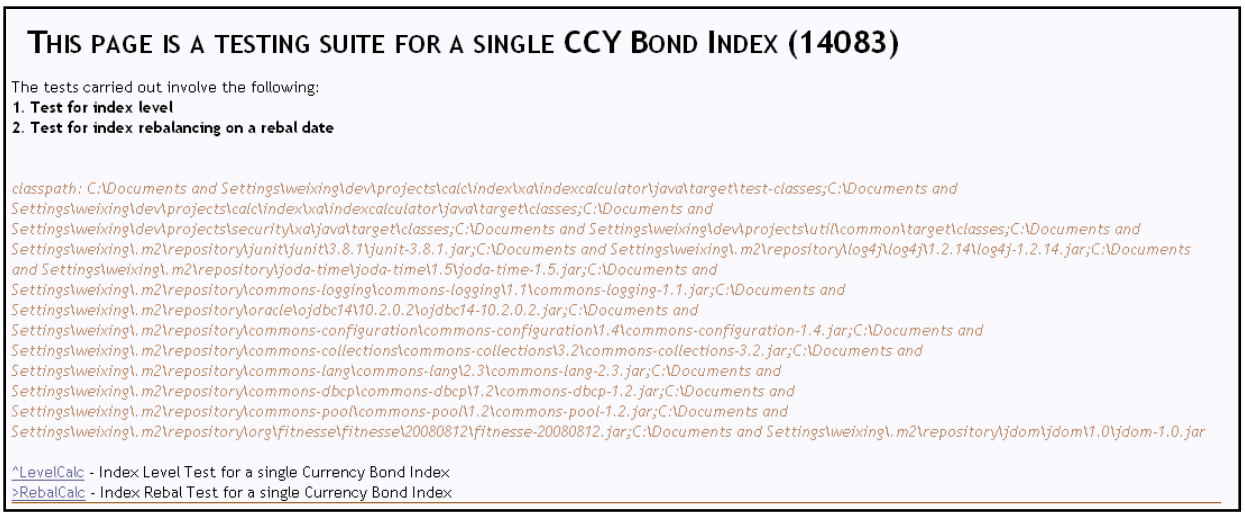


Figure 17 - Test suites for specific asset types

5 Integrating FitNesse with Hudson

5.1 Introduction and Problem Definition

Though FitNesse provides an easy to use interface, it is not ideal on its own for automated testing. Running FitNesse as-is still requires a tester to go to the server, and run every test or test suite to make sure that the changes didn't affect the tests. To automate FitNesse testing, we decided to integrate our FitNesse tests with the Hudson build system.

5.2 Implementation

Maven and Fitnesse users had already created a plugin to use for such integration. As stated above, Maven uses POM files for building projects. We used the provided POM file with some changes to get our tests to run – namely, we directed the POM file to point to our FitNesse server and test page. Once we properly configured the file, we just needed to enter mvn fitnesse:run to run our tests. This call automatically ran our tests and produced a report with the results.

Once we had FitNesse integrated with Maven, it was a simple transition to integrate Fitnesse into Hudson. We went into our original Hudson configuration and added a new job for this test project. This project used the same POM file as the original Index Calculator project with the fitnesse:run Maven goal specified. Hudson printed the FitNesse tests results both to the console and to a new HTML page. Figure 18 and Figure 19 below show these results.

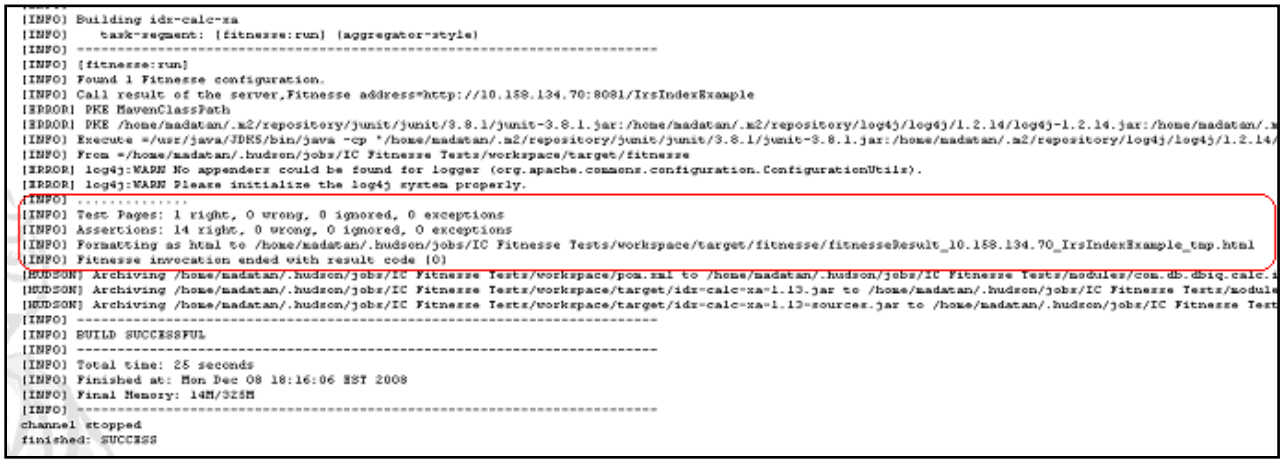


Figure 18 - Hudson console output after running FitNesse tests

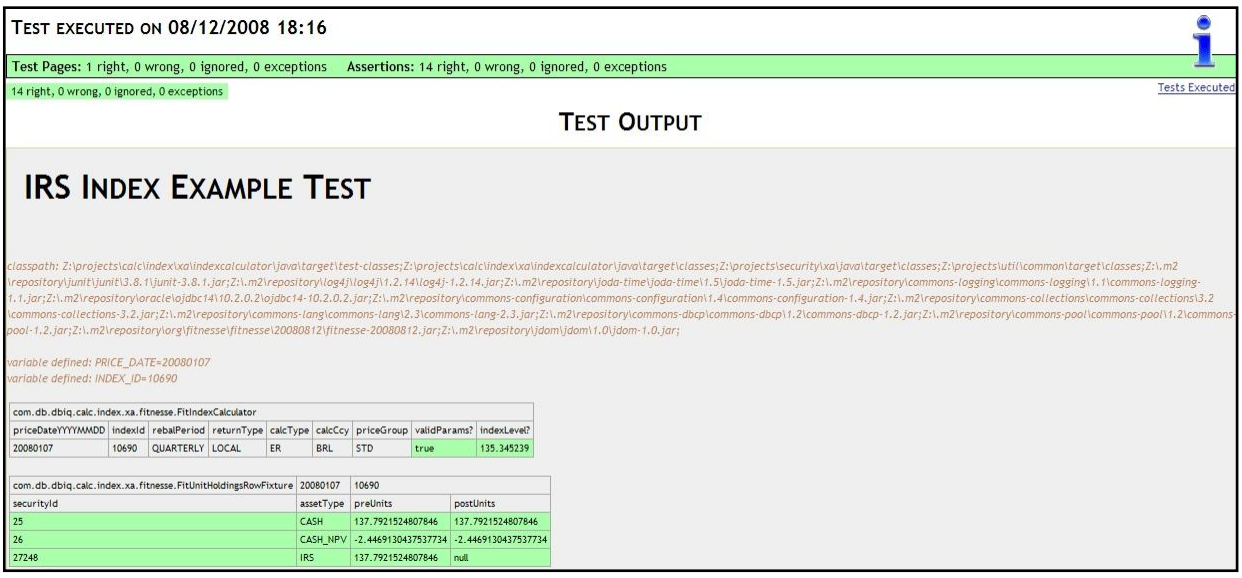


Figure 19 – HTML page output by FitNesse through Hudson

5.3 Results

We then had a Hudson instance with two related projects: the FitNesse tests for the Index Calculator and the calculator itself. To get both of the jobs to run together, we had to change a simple setting in the IC project to automatically run the FitNesse tests after the IC passes. However, the tests will only run if the IC passes and is stable, that is, every JUnit test must also pass. This is in general a good feature because acceptance tests are meant to be an addition, not replacement, for unit tests. We worked around this by creating another new job that builds the IC but does not run the JUnit tests. We did this by changing our call to the Maven file to install while skipping the tests. Using this job demonstrates that the FitNesse tests runs automatically, but only under the right conditions. Figure 20 shows this successful execution. Until all of the JUnit problems are solved, Deutsche Bank is left with the choice of executing the JUnit tests or the FitNesse tests automatically.

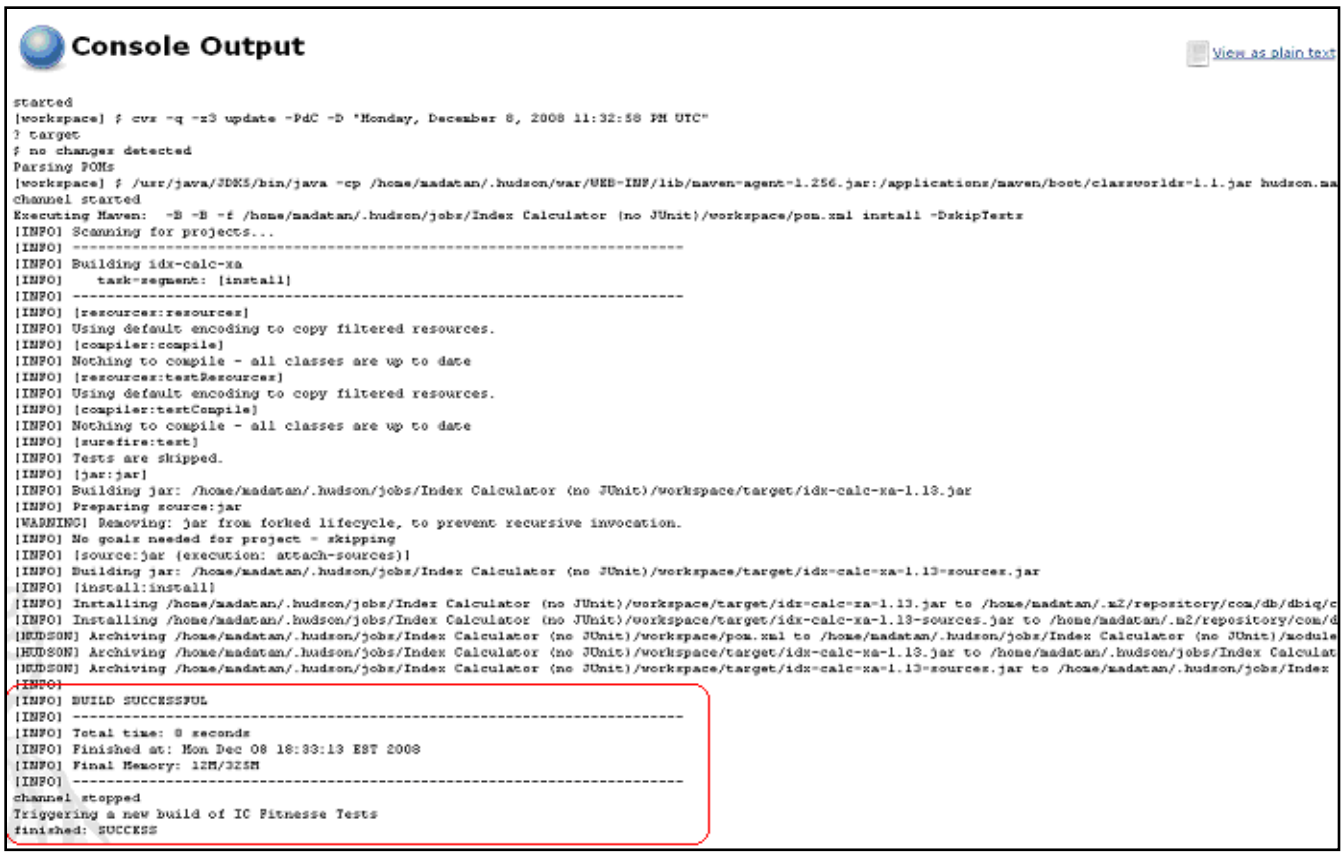


Figure 20 – Successful builds of both the Index Calculator and its FitNesse tests

6 Conclusion and Recommendations

With the integrated build and testing system successfully deployed, the DBIQ team now has a single source to monitor various projects along with their corresponding acceptance tests. This is not only useful for a developer who can now easily keep an eye on the build process and receive notifications, but it is also very useful for the business end users. With the deployment of acceptance tests through FitNesse, they can easily validate results without having to understand or even look at the code. Such a tool did not exist before and is a new and easy way for DBIQ to bridge the gap between the development and business side of DBIQ.

Although these tools are useful and their deployment has been successful, there is a much greater potential still untapped. We wish to expand on these areas where these tools can be exploited even further to provide great benefit not only to DBIQ but to Deutsche Bank in general.

Firstly, we wish to address an issue that is DBIQ specific. The current testing spectrum of FitNesse at DBIQ is somewhat limited and does not cover each and every asset type. Different asset types carry out calculations in different ways and hence require different input and provide different results. To verify the functionality of the IC, it is important that each and every asset type is covered so that nothing is left to chance. With some understanding of the various asset types and how they use the IC, this process can be achieved without much difficulty and will certainly be extremely useful.

Secondly, FitNesse is a tool that can provide a business end user with an easy way to verify and understand tests. Keeping in mind that Deutsche Bank is a financial services firm, the IT teams are providing support or applications to a corresponding financial/business team. A tool such as FitNesse can easily be used in other such departments as a means to improve the understanding and communication between the IT and business teams.

We also have some recommendations on the application of Hudson. The functionality and use of Hudson can be extended to upon a great extent. With the usage of the email notification tool in Hudson, various project teams can be contacted when builds fail/succeed. There are ways of customizing this process so that someone closely involved with a project is receiving all notifications where as someone with a smaller role could only be contacted once a week. Hudson not only makes it easy to manage projects but with development of most projects taking place all over the world – benefits such as

notifications would make it much easier for those involved to catch the errors early rather than wondering days, weeks, or even months later what went wrong. Another advantage is that regarding related projects, one can be set to trigger the build of the other. As a result, if Deutsche Bank decides to use this tool on a much larger scale, it would certainly help in organizing various dependent projects and their results together. This tool is a potential time and risk saver and we recommend that its use be expanded to as many IT departments as possible.

Finally, combining the tools of Hudson and FitNesse is extremely useful. It provides teams with easy monitoring and validating methods. As mentioned above, we feel that these can find a place in various IT departments not only as separate tools but also as an integrated system which could be useful for all involved in the project.

7 Appendix A

7.1 Testing Tools

We took a look at some of the tools out there for testing purposes and have come up with a list of 3 which could potentially be useful to DBIQ. A short description of each tool is mentioned and it is discussed in comparison to FitNesse as well.

7.1.1 JFunc[10]

JFunc is a solution for functional (acceptance) tests that extends the JUnit framework. JUnit is an open-source project supported by Terraspring, Inc. JFunc addresses the differences between unit and acceptance tests, namely, the fact that unit tests should be able to run in any order whereas functional tests require a specific sequence. Also, it gives testers the capability to decide whether the testing should halt on failure or continue running the rest of the tests.

JFunc is geared towards software developers, unlike FitNesse, which is designed to be used by any business user. JFunc requires knowledge of Java and JUnit, where FitNesse only requires parameter and function names. FitNesse is easier also in that it is wiki-style and anybody who needs the information should easily be able to find the page and tests they are looking for. JFunc is more standard, especially if there are already JUnit tests present for a project. Also, while FitNesse has a simple “Test” button to run all of a class’s tests, IDE’s such as Eclipse have functions that will simply run all of the JUnit and JFunc tests at once. For a system such as DBIQ where the only people testing the software are developers who understand Java and JUnit, JFunc may be simpler and the better tool to use for the job.

7.1.2 DDSteps[11]

DDSteps also extends on the JUnit framework for the purpose of data driven test cases. In short, DDSteps lets you parameterize your test cases, and run them more than once using different data. This

[10] (JFunc)
[11] (DDSteps)

can in some sense be compared to row and column fixtures in FitNesse and provides a similar advantage.

DDSteps seems really easy to install and get started – it only needs to be added to the classpath. DDSteps uses external test data (in Excel) which is injected into the test case using standard JavaBeans properties. The test case is run once for each row of data, so adding new tests is just a matter of adding a row of data in Excel.

DDSteps seems like a tool for someone who does not need to much about the code once a test case is set up. One can easily extend the test by simply using the Excel spreadsheet. However, FitNesse does a better job of alienating the code from someone who just wants to run and extend some test cases. Nevertheless, for a developer, this could be a useful tool.

7.1.3 Concordion[12]

Concordion is a testing framework for java code similar to FitNesse. Concordion provides users with GUI interface on a web browser, and tests can be run easily after parameters are defined. It is very easy to learn and use. HTML files are the primary documents used to control and manage projects in Concordion. Table 1 illustrates a comparison between FitNesse and Concordion.

Table 1 - FitNesse vs. Concordion

	FitNesse	Concordion
Mapping document contents to Java code	Heading rows in tables form implicit mappings. **Pros:** Can work with HTML or with Excel spreadsheets.	Explicit (but hidden) instrumentation in the document performs the mapping. **Pros:** Fixture code is clean and simple. Obvious what is called and when.

[12] (Concordion)

	Cons: It can be hard to see which table maps to which class / method.	**Cons:** Requires an HTML source document.
Storage of specifications	Wiki	In a Java source folder - can be stored under version control with the rest of the code.
Integration	Separate runner	Run using JUnit

It seems difficult to pick a winner amongst these two as they both seem very comparable. Noting this, it isn't surprising that the developers of Concordion were originally inspired by the Fit Framework. We feel that this is a tool that can be further investigated as a possible recommendation for Deutsche Bank.

8 Appendix B – Weekly Reports

8.1 Weekly Report 1

Problem Definition:

The DBIQ team at Deutsche Bank wishes to establish a continuous integration system for their code base. This allows the advantage to build/compile the code at any time. This makes it easier for developers to integrate changes to the project, and easier for users to obtain a fresh build. A tool such as this can increase efficiency and productivity.

Goals accomplished this week:

Establishment of Hudson

Integration of Hudson with Maven

Deployment of FitNesse

Establishment of Hudson- an extensible continuous integration system

The first step of the project was to establish Hudson in a UNIX environment for the purpose of deploying an extensible continuous integration system. We started off finding resources on the internet on Hudson. The official page of Hudson (http://hudson.dev.java.net) was a primary resource we used. After downloading and setting up the latest version of Hudson, we were able to successfully launch it.

Integration of Hudson with Maven

Since the code base we are dealing with uses Maven as a build framework, the step next was to integrate Hudson with Maven and the existing code. To test if this integration works, we checked out projects that ran successfully in Maven and executed these projects in Hudson. During the building process, we encountered and resolved problems such as adjusting configurations, adding dependencies,

and importing missing modules. In the end, we were able to successfully build majority of the projects with Hudson, confirming the establishment and functionality of Hudson deployed.

Deployment of FitNesse

FitNesse is a wiki-style web server used to collaboratively test software. We used the official fitnesse user's guide located at http://fitnesse.org/FitNesse.UserGuide to deploy the server on both a local windows machine and the remote Linux server. Though we successfully installed the tool, we will need to do more research before we can begin using it to run the existing tests.

Tentative Goals for next week:

Further investigation on FitNesse and its use.

Start extending DTM for each asset type.

8.2 Weekly Report 2

Jenny, a member of the DBIQ team is London will be working with us on the rest of our project and is currently in the process of carving the exact details out as well as giving as preliminary assignments to get us started. Mich, another member of the DBIQ team, is also working with us on the project and gave us some beginning assignments too.

Goals accomplished this week:

Further investigation on FitNesse and its use.

Understanding the high level functioning of the Index Calculator.

FitNesse Investigation

Some other people in the company who have worked with FitNesse sent us links to their wiki pages containing tests. We ran the tests and looked at the code behind the wiki to see how it worked. Jenny then assigned us to set up our own local copy of FitNesse and run a test case on it to understand its use and implications. We did so using her test page as a basis, learning how each page interacts with the Java code.

SQL & SQL Developer

After learning about SQL on our own, we got help with the specific system used here.

Mich created an assignment for us in regards to the different types of securities that go into index calculations in order to get a better understanding of the system.

We each went through five of the views that represent different types of assets, and saw the interaction of the tables involved in each view. This gave us a better understanding of the specific database system used here.

Investigation of other Testing Tools

While the team is trying to come up with our precise goals and get the ball rolling, we were asked to look into some alternative testing tools with functionality similar to FitNesse. Three tools that caught our attention were JFunc, DDSteps and Concordion. A document was written up expressing our understanding of how to use these tools.

Index Calculator

To be able to write test cases properly, we need to understand the Index Calculator, the main class that this entire project is based off of. The Java code for this class was somewhat confusing, but we looked through it, followed the method calls, and got a decent understanding of how it worked. In one reference document that Jenny sent us, there was a flow diagram describing how the calculator worked, which helped us to at least follow along with the calculator's algorithm.

Hindrances

This week we faced some typical issues that hinder new hires! The first couple of days in the week were spent getting things set up, obtaining access and logistical issues, etc. However, we seem to be all set with everything that is needed and are ready to get the ball rolling.

Tentative Goals for next week:

Unit test fixture development and implementation.

Integration of further developments with FitNesse.

8.3 Weekly Report 3

Goals accomplished this week:

Write customized code for various assets types to run advanced tests on FitNesse.

Override default fixture to display difference between expected and actual results.

FitNesse Test on Index Rebalancing

Jenny wanted us to build a new test dealing with index rebalancing. Rebalancing takes place periodically in the life of an index, in which some new securities are added and some old securities are removed, and the weights of each components of the index are redistributed. Testing index rebalancing is more complex than tests we wrote previously. In addition to regular properties such as Security ID, Currency and Price, advanced properties like Percent Current Weight, Percent Target Weight, Units Pre-rebalancing and Units Post-rebalancing are all taken into consideration for the test.

We first extracted the data from database that contains the information required for the tests. And then we compiled and integrated JUnit tests with the FitNesse. This test was different from the ones we wrote last week because we wrote the FitNesse fixture (the java code to go along with the wiki page) ourselves. We used the other fixtures as a basis for this, but ran into problems because it took some time to understand what the calculator was actually doing and how each of the objects involved interacts with the others. Eventually we got through this and ran the test successfully. The entire process was completed in time and Jenny seemed very happy with the work.

Display of Difference between Expected and Actual Results

The default method FitNesse to display an unexpected result after the test is to show the expected result and actual result in a red cell, whereas the expected result is displayed in a green cell. One feature that would be useful to us is to have the difference between the actual value and expected value displayed, since it would be useful to people who analyze the data.

To achieve this, we started off researching different approaches: use wiki markup or override default row fixture. After reading user guides and examples on FitNesse website, we concluded this feature cannot be established solely with wiki markup. In FitNesse, whenever a test comes up wrong, it changes color and the table cell displays both the actual and expected values. We figured that we need to do something similar to this, but add a third value in the cell of the delta. This took just overriding one function, wrong(), which is called whenever the answer comes out wrong.

Jenny wanted the deltas to be displayed in new columns, but this was a much harder task. We wanted to create a new object including the delta values, but could not manipulate our data after the program reports which cells were wrong. After a lot of searching, we found that we could create a blank column, and override the handleBlankCell() method to display a value. By saving delta as a global variable that changes every time either wrong() or right() is called (to compute the new delta, whether it be zero or not) we could put a new value in each blank cell and display as Jenny wanted. Once again, she seemed very pleased with the work.

<u>Tentative Goals for next week:</u>

Our next step for this Index Rebalancing test is to change our tests to run the calculators themselves rather than just retrieve the already calculated values from the database. To do this we will need to call the calculators, write data to the database and then delete it as well once the test is carried out. Work on this has begun on this and we are going hand in hand with Jenny to carve the details out.

Also, Jason might possibly be adding another component/task – we will update you on this once we have more information.

8.4 Weekly Report 4

Goals accomplished this week:

Addition of test suite, modifications in FitNesse to make it easier to understand for a business end user.

Add and run more tests of different indices on FitNesse.

Investigate and develop automatic data loading in FitNesse.

Generating SQL queries for inserting data required for the tests into the database.

Sanity checks and Metadata

FitNesse Modifications and Test Suite

Given that FitNesse had already been set up and running as expected, we spent some time this week trying to further modify the look and feel of FitNesse. This included making more changes to the way results are displayed, changing the home page to present all the test cases in a more organized fashion and other small things which would make using FitNesse extremely easy to understand and comprehend. We also created a 'test suite'. This is simply a page which can combine a certain number of tests and allow the user to examine the results on just one page. It helps in organizing tests and can be of great use as the number of test cases continues to grow.

FitNesse Test on Multi-currency Index

After successful execution of Index Rebalancing test last week, we continue our implementation and investigation of FitNesse by adding another test. This week, we started looking into a multi-currency index. Although the member securities are their attributes were different from the test done last week, the test results were of the same type. This new test was created and is successfully running.

Data loading in FitNesse

Right now the FitNesse tests are calling on the code that queries the database for existing data. This data is then returned to FitNesse, which checks the data against the expected result. Since we got our

expected values from the database, these will always return true! What we wish to do is have FitNesse pass in input insert queries that can create the data required for the tests in the database and then run the calculator to check against these values. We have started working on this and currently we have reached a stage where we can carry out insert queries from FitNesse to write to the database.

SQL Insert Query Generation

In order to achieve the task mentioned above, we need to generate the Insert statements from SQL and then put those through on FitNesse. We spent some time this week generating these insert statements for the test we were dealing with. These involved 2 particular indices and all their member securities along with their attributes.

Sanity Checks and Metadata

This Thursday, 2 new projects were added to our domain. One of these includes 'sanity checks'. The projects involves a lot of analysis on the current sanity check definitions, sanity check maps and on extracting useful information on which checks fail the most and so on. The purpose of the project is to have a greater understanding on the large sanity check framework at DB in order to possibly recommend where improvements and reductions could be made in this area.

The 2nd project involves 'metadata'. Metadata is like an execution framework which tells the Index Calculator Engine what to do and when. The metadata has only been developed for a couple of indices so far and the project involves developing the metadata for about another 100 indices.

Tentative Goals for next week:

Further development of writing to the database and carrying out tests that do so.

Further analysis on sanity checks and continuation of developing the metadata.

8.5 Weekly Report 5

Goals accomplished this week:

1. Adding more test cases of new asset types to FitNesse.
2. Installing Maven plug-in to interact with FitNesse.
3. Returning to Hudson continuous integration system and calling FitNesse from Hudson.

New Test Cases

Since we have been able to proficiently set up tests on FitNesse, we put up six more test cases. These were different from the previous two test cases because several tables associated with the index calculators being tested in these cases are missing or incomplete. However, we were provided with the excel sheets which contain the algorithms of calculating the level values of corresponding indices. After studying and understanding the excel sheets, we found the information needed to rebuild the data tables required for the tests. So we populated the tables with correct data and ran these tests. We also modified the appearance and organization of test cases on web testing interface. We created a table on the front page with description and index ID placed next to the test links. We also organized the tests into suites – one suite for each index ID with two tests, and one overall suite to run tests for every ID at once.

FitNesse – Maven Plug-in

One feature that can be very useful in addition to our system is to run FitNesse test through Maven. We found an open-source Maven plugin to run FitNesse tests. We had a lot of trouble setting up the Project Object Model (pom.xml) file. In the pom file, we had to ensure that the classpath was correct, and that we properly specified the FitNesse server and page. We had trouble with some of these details, but with help from our sponsors and a lot of guesswork we eventually properly configured the plugin.

Calling FitNesse from Hudson

Since we had our Maven plugin to call on our FitNesse case, we decided to take the project to the next level and integrate FitNesse with Hudson. We returned to our original Hudson installation and ensured that we had a project created to compile (and run the JUnit tests for) the Index Calculator. We created a new job to run the FitNesse tests through Maven, and set Hudson to run this job whenever the Index

Calculator build runs. We ran into an issue because some of the JUnit tests for the Index Calculator fail, causing an unstable build, preventing Hudson to continue on with the FitNesse tests. However, after talking to our sponsors, we decided that this was desired behavior – why run acceptance tests if the unit tests fail? We decided to leave Hudson as it was here, and allow the FitNesse tests to only be run as a separate job or if the Index Calculator JUnit tests all pass.

Tentative Goals for next week:

1. Migration of final Fitnesse system into new test database.
2. Migration of work from our local machines to a central server
3. Finish staging the data for our test cases
4. Finish the paper and presentation!!
5. Completion of the entire project by delivering a self-contained user-friendly testing system for index calculators.

8.6 Weekly Report 6

Goals accomplished this week:

1. Adding more test cases of new asset types to FitNesse.
2. Adding a deletion method to each fixture.
3. Expanding RebalCalc Tests.
4. Finalizing the project by putting every component together.
5. Completing final report.

New Test Cases

Jenny had two new indices that we added to our testing system. Both of them are money market indices, distinguished by the existence of spreads. Thanks to our previous experience, we managed to put them up on FitNesse and then execute the tests quickly and successfully.

Deletion Method / Teardown

In order to establish a complete standalone testing system, we also needed to delete the data used after a test is run. During this week, with Jenny's example code, we extended deletion method to each data handler fixture we have written. So our current FitNesse tests are completely standalone – they insert data required for a specific test into the database, execute the test, and then delete the data just inserted to keep the database clean.

RebalCalc Test

Each index test case we have built on FitNesse consists of two individual tests: LevelCalc Test and RebalCalc Test. During the past several weeks, we have been mainly dealing with LeveCalc, which tests the functionality of the index calculator by running it with historic data and checking for generated index levels. The rebal calculations deal with dates when member securities change or weights change, etc. Some new tests were added to verify calculations on rebal dates.

Hudson and FitNesse

As mentioned last week, we were able to integrate the tools Hudson and FitNesse. We spent a little more time examining and building on the same this week.

9 Appendix C - Additional Tasks

In the midst our work towards accomplishing the final goal of having a continuous integration and testing system, some additional tasks were also assigned to us. These tasks, though not directly related to our overall goal, were seen as measures important in improving the overall quality of the work on the development side here at DBIQ.

9.1 Sanity Checks

The DBIQ team is responsible for ensuring that each index calculator is processed properly and normally by the calculation engine so that the indices can be published without any errors. The method DBIQ team uses is to program individual sanity checks for different indices. Developers will be notified of the results of each sanity check and then fix the problem. However, since Deutsche Bank manages and maintains thousands of indices, there are more than 1,600 defined sanity checks currently being used. This large amount of sanity checks creates difficulties for developers to identify, manage and respond to each failed check accurately and efficiently. Our task was to categorize all the sanity checks available with respect to its functionality, and then organize the severity of failed checks by looking at historical data.

The team manages sanity checks by tables in Oracle database. A table called “sanity_chk_defs” stores the definitions of each sanity check, including the check ID, description, code name, parameter list and version among others. However, a particular check and be used several times. Hence, another table “sanity_chk_map” contains the detailed records of all the checks being used. Under our director’s instruction, we divided the checks by their functions into thirteen categories. By looking at historic data during the previous two weeks, we summarized and calculated the percentage of failed checks of each category, illustrated in the pie chart below.

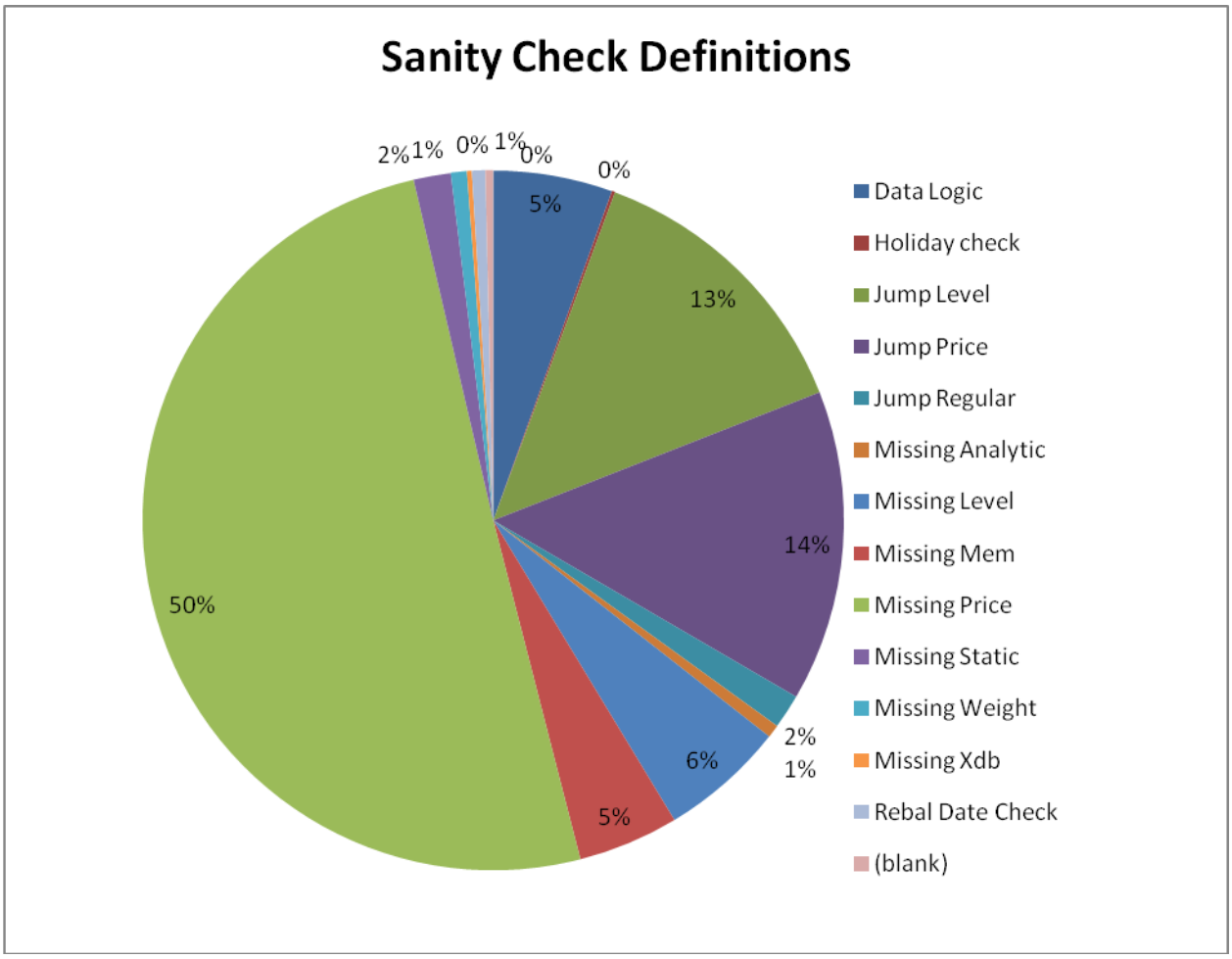


Figure 21 – Sanity check definitions failing by category

Figure 21 helps the developers to identify the severity of each category of failed checks. Furthermore, it answers many questions that can guide developers to efficiently use and manage the sanity check system in the future.

Another part of sanity checks that we looked into was the family into which the checks belong. Each definition of a sanity check is assigned a family based on what type of structure it carries out a test on. To do this, about 16000 maps needed to be analyzed based on their families and then organized based on their failure rate.

Table 2 – Excerpt of Sanity checks failing by family

FAMILY	FAIL	WARN	TOTAL
Grand Total	**14834**	**2872**	**17706**
XAlpha	767		767
DBAIMS	333	399	732
XALPHA_SPIIEUR_INDEXPAIR	727		727
XALPHA_SPIIIEUR_INDEXPAIR	726		726
XALPHA_SPIIIUSD_INDEXPAIR	726		726
XALPHA_SPIIUSD_INDEXPAIR	726		726
XALPHA_SPEUR_INDEXPAIR	722		722
XALPHA_SPUSD_INDEXPAIR	722		722
EMLC	65	428	493
Standard Equity Indices	326	165	491
High Yield	287	37	324
iBoxx Euro	191	108	299
Asian FRB	297		297
USD Liquid Credit	258		258
APINDEX	256		256

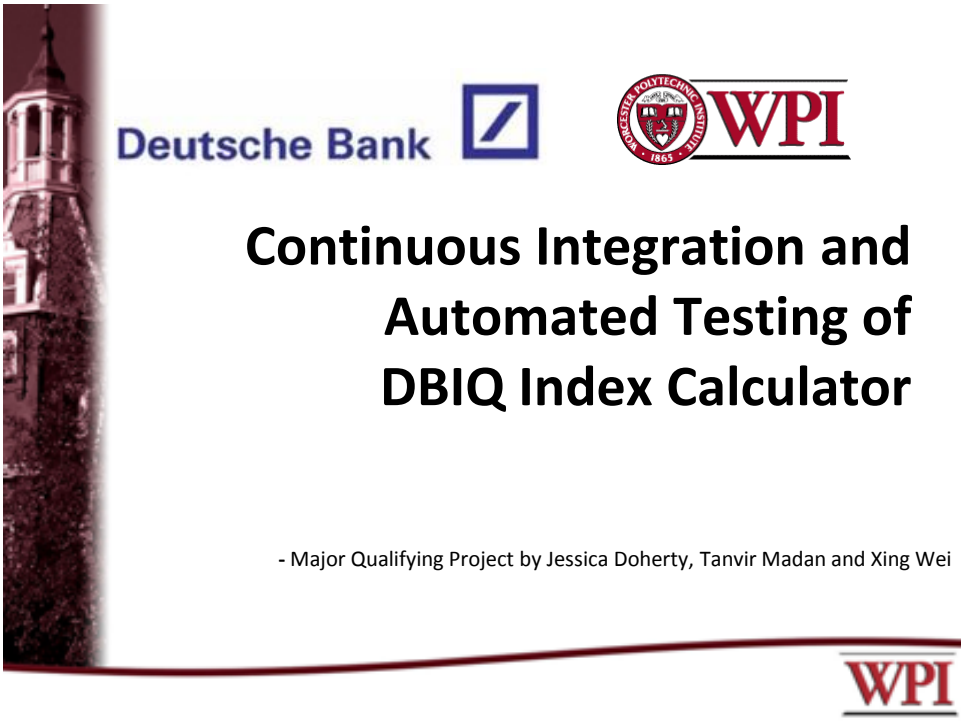
This project only lasted a couple of days as focus went back to the main goal of continuous integration and testing. However, it is something that Deutsche Bank wishes to pursue more in the future. Not only will this help them get a deeper understanding of which checks are failing and why, it could also lead to recommendations on how to improve the sanity check framework currently being used.

9.2 Metadata

Metadata can be described simply as “Data about data”. It is used to describe what a particular piece or set of data means. One set of such data that DBIQ keeps track of is the execution sequence for the index calculator. This is stored in a database table (dbiq_sequence) by the index ID, each ID having a set of calculation steps which either stage data or call an execute function. The steps which involve calling an execute function map to another database table (dbiq_execute). For each step, that table contains the class which needs to be executed and the parameters to pass into that class. However, being nearly empty, these metadata tables have little function.

The DBIQ team had the metadata tables set up only some a few indices and wanted to get them set up for all indices. To populate these data tables, we of course used SQL INSERT statements. We first filled the dbiq_sequence table for each of the index IDs given to us. This involved getting the index IDs by asset type and setting each step by which level to calculate or which step to execute. We then used the execution steps that we added to populate dbiq_execute. However, the information regarding that data is spread throughout DBIQ and we could only fill the execution class and parameter for one of the twenty asset types. With the information that we filled these metadata tables DBIQ cannot yet extract all of the information they will need, but we have at least provided the framework for completely filling in the information.

10 Appendix D – Presentation



Introduction

- The Deutsche Bank Index Quant (DBIQ) team has developed and uses an Index Calculator (IC) to calculate analytics in relation to various benchmark and tradable indices.

- Since the IC is subject to constant change, validating its functionality is of utmost importance.

WPI

Objectives

Our task was to implement tools to reduce business risks by identifying possible errors resulting from code changes.

- Continuous Integration: Allows to catch failures early to quicken the development process and reduce risks.
- Automated Regression Testing: Allows to easily run and understand tests for the entire spectrum of the Index Calculator.

Methods

- Continuous Integration (Hudson) – Rebuild the IC automatically every time a change is made.
- Automated Testing (FitNesse) – Develop and run reproducible regression tests that are easy for business users to understand and extend upon.
- Comprehensive Build System – Automatically run the tests after running the build.

Hudson

An Easy-to-Deploy Continuous Integration System

WPI

Hudson: Purpose

- Once deployed, can be used and configured by anyone at anytime simply from a website.
- Allows for immediate reporting of any issues that arise.
- Projects can be automatically built when the code is updated or at a specific time.
- One project can be set to trigger the build of another.

WPI

Hudson: Our Role

We deployed and configured Hudson, which consisted of:

- Directing Hudson to download the newest project code from the DBIQ repository.
- Setting Hudson to use the existing Maven build file and to execute the goals of the file whenever a change is made in CVS.

WPI

Hudson: Current Status

- Hudson is deployed and set to automatically build the Index Calculator whenever a change is made in the repository.
- Anyone with the URL can enter Hudson and run the build as they desire.
- Other projects can be added to be built in this Hudson deployment.

WPI

FitNesse

Automated Testing for the Index Calculator

WPI

FitNesse: Purpose

- Once deployed, FitNesse can be accessed by anyone through a simple website.
- Acceptance tests are created in a wiki-style format that are easy to understand and easy to create – very useful for business end users.
- Organize tests into test suites that can be run with one click.

WPI

FitNesse

```
!|com.db.dbiq.calc.index.xa.fitnesse.FitUnitHoldingsRowFixture|${PRICE_DATE}|${INDEX_ID}|
|securityId|assetType|preUnits|postUnits|
|25|CASH|137.7921524807846|137.7921524807846|
|26|CASH_NPV|-2.4469130437537734|-2.4469130437537734|
|27248|IRS|137.707846|null|
```

com.db.dbiq.calc.index.xa.fitnesse.FitUnitHoldingsRowFixture	20080107	10690	
securityId	assetType	preUnits	postUnits
25	CASH	137.7921524807846	137.7921524807846
26	CASH_NPV	-2.4469130437537734	-2.4469130437537734
27248	IRS	137.707846 *expected* 137.7921524807846 *actual*	null

WPI

FitNesse: Our Role

We deployed FitNesse and wrote tests for the Index Calculator, which involved:

- Writing test tables in wiki-format.
- Developing new Java fixtures to run the test cases.
- Writing new classes to override FitNesse defaults and reorganizing the fixtures to extend these.
- Creating new data types and extending a new database access class to insert and delete from the test database.

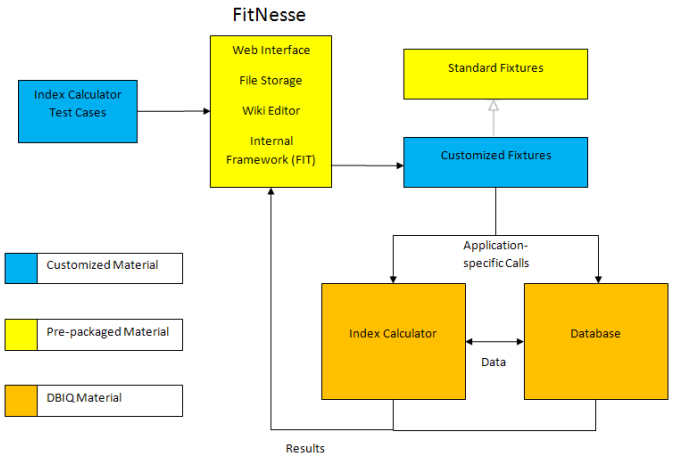
WPI

FitNesse: Current Status

- New Java fixtures test various asset types.
- New acceptance tests relying on these fixtures on the FitNesse wiki.
- Twenty tests currently cover level and rebal calculations for certain asset types.
- Tests are self-contained with a set up and tear down inserting and removing all required data.

WPI

FitNesse and DBIQ



WPI

Final Product

Comprehensive Build System

WPI

Comprehensive Build System

- Hudson currently builds the Index Calculator whenever a change is made in the repository.
- FitNesse currently provides regression testing for the IC.
- Testing is essential whenever a change is made to the repository – so our build system should do so.

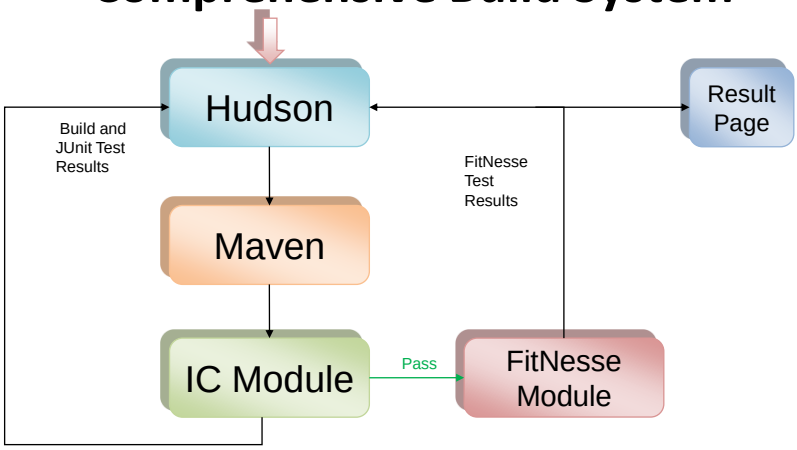
WPI

Integrating FitNesse with Maven

- Found and deployed an open-source Maven plug-in to interact with FitNesse.
- Hudson now first builds the Index Calculator and then runs the FitNesse tests every time a change is made to the repository.

WPI

Comprehensive Build System



WPI

Demo

WPI

Recommendations

- Extend our FitNesse framework to include all asset types.
- Add new Hudson jobs to automatically build other projects and run related tests.
- Extend use of these integration tools to other departments.

WPI

Deutsche Bank

WPI

Acknowledgements

We would like to extend our thanks to our sponsors and advisors who have provided constant help and support throughout the project.

WPI

11 References

Automated Testing. (n.d.). Retrieved from http://www.exforsys.com/tutorials/testing/automated-testing-advantages-disadvantages-and-guidelines.html

Concordion. (n.d.). Retrieved from http://www.concordion.org/ScriptingMakeover.html)

DBIQ Index Calculator. (n.d.). Retrieved from http://nyiqappp1.us.db.com:8668/space/Projects/DBIQ+Rearchitecture/Index+Calculators/Index+Calculator.bmp

DBIQ Wiki/Blog. (n.d.). Retrieved from http://nyiqappp1.us.db.com:8668/space/Projects/DBIQ+Rearchitecture/Index+Calculators

DDSteps. (n.d.). Retrieved from http://www.ddsteps.org/display/www/DDSteps)

FitNesse. (n.d.). Retrieved from http://fitnesse.org/

Hudson. (n.d.). Retrieved from https://hudson.dev.java.net/

JFunc. (n.d.). Retrieved from http://jfunc.sourceforge.net/

JUnit. (n.d.). Retrieved from http://www.junit.org/

Maven. (n.d.). Retrieved from http://maven.apache.org/